Volume 21 Number 4

ACTA CYBERNETICA

Szeged, 2014

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed. There are no page charges. An electronic version of the puplished paper is provided for the authors in PDF format.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements:

- title of the paper
- author name(s) and affiliation
- name, address and email of the corresponding author
- An abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.). Manuscripts must be submitted by email as a single attachment to either the most competent Editor, the Managing Editor, or the Editor-in-Chief. In addition, your email has to contain the information appearing on the title page as plain ASCII text. When your paper is accepted for publication, you will be asked to send the complete electronic version of your manuscript to the Managing Editor. For technical reasons we can only accept files in LaTeX format.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: acta@inf.u-szeged.hu

Web access. The above informations along with the contents of past issues are available at the Acta Cybernetica homepage http://www.inf.u-szeged.hu/actacybernetica/ .

Ferenc Gécseg
Department of Computer Algorithms
and Artificial Intelligence
University of Szeged
Szeged, Hungary
gecseg@inf.u-szeged.hu

Jozef Gruska
Institute of Informatics/Mathematics
Slovak Academy of Science
Bratislava, Slovakia
gruska@savba.sk

Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

Helmut Jürgensen
Department of Computer Science
Middlesex College
The University of Western Ontario
London, Canada
helmut@csd.uwo.ca

Zoltan Kato
Department of Image Processing
and Computer Graphics
Szeged, Hungary
kato@inf.u-szeged.hu

Alice Kelemenová
Institute of Computer Science
Silesian University at Opava
Opava, Czech Republic
Alica.Kelemenova@fpf.slu.cz

László Lovász
Department of Computer Science
Eötvös Loránd University
Budapest, Hungary
lovasz@cs.elte.hu

Gheorghe Păun
Institute of Mathematics of the
Romanian Academy
Bucharest, Romania
George.Paun@imar.ro

András Prékopa
Department of Operations Research
Eötvös Loránd University
Budapest, Hungary
prekopa@cs.elte.hu

Arto Salomaa
Department of Mathematics
University of Turku
Turku, Finland
asalomaa@utu.fi

László Varga
Department of Software Technology
and Methodology
Eötvös Loránd University
Budapest, Hungary
varga@ludens.elte.hu

Heiko Vogler
Department of Computer Science
Dresden University of Technology
Dresden, Germany
Heiko.Vogler@tu-dresden.de

Gerhard J. Woeginger
Department of Mathematics and
Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
gwoegi@win.tue.nl

Ferenc Gécseg

(1939-2014)

We regret to inform you that Ferenc Gécseg, former Editor-in-chief of Acta Cybernetica died on October 6, 2014. We will publish a separate issue to his memory.

Acta Cybernetica **21** (2014) 507–527.

Quotient Complexity of Bifix-, Factor-, and Subword-free Regular Languages*

Janusz Brzozowski[†], Galina Jirásková[‡], Baiyu Li[†], and Joshua Smith[†]

Abstract

A language L is prefix-free if whenever words u and v are in L and u is a prefix of v, then $u = v$. Suffix-, factor-, and subword-free languages are defined similarly, where by "subword" we mean "subsequence", and a language is bifix-free if it is both prefix- and suffix-free. These languages have important applications in coding theory.

The quotient complexity of an operation on regular languages is defined as the number of left quotients of the result of the operation as a function of the numbers of left quotients of the operands. The quotient complexity of a regular language is the same as its state complexity, which is the number of states in the complete minimal deterministic finite automaton accepting the language.

The state/quotient complexity of operations in the classes of prefix- and suffix-free languages has been studied before. Here, we study the complexity of operations in the classes of bifix-, factor-, and subword-free languages. We find tight upper bounds on the quotient complexity of intersection, union, difference, symmetric difference, concatenation, star, and reversal in these three classes of languages.

Keywords: bifix-free, factor-free, finite automaton, quotient complexity, regular language, state complexity, subword-free, tight upper bound

1 Introduction

The state complexity of a regular language L is the number of states in a minimal deterministic finite automaton (dfa) accepting L [41]. This complexity is the same as the quotient complexity [5] of L, which is the number of distinct left quotients of L. We prefer quotient complexity since it is more closely related to properties of languages. The quotient complexity of an operation in a class $\mathcal{C}$ of regular languages is the maximal quotient

*This work was supported by the Natural Sciences and Engineering Research Council of Canada under grant no. OGP0000871, by the Slovak Research and Development Agency under contract APVV-0035-10 "Algorithms, Automata, and Discrete Data Structures", and by VEGA grant 2/0183/11.

[†]David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada N2L 3G1. Baiyu Li's present address: Optumsoft, Inc., 275 Middlefield Rd, Suite 210, Menlo Park, CA 94025, USA. Joshua Smith's present address: Spielo International Canada ULC 328 Urquhart Ave. Moncton NB, Canada E1H 2R6. E-mail: {brzozo,b5li,}@uwaterloo.ca, belowtwenty@hotmail.com

[‡]Mathematical Institute, Slovak Academy of Sciences, Grešákova 6, 040 01 Košice, Slovakia. E-mail: jiraskov@saske.sk

complexity of the language resulting from the operation, taken as a function of the quotient complexities of the operands in class $\mathcal{C}$. For more information on state and quotient complexity see [5, 6, 41].

One of the first results concerning the state complexity of an operation is the 1966 theorem by Mirkin [33], who showed that the bound 2^n for the reversal of an n-state dfa can be attained. In 1970 Maslov [32] stated without proof the bounds on the complexities of union, concatenation, star, and several other operations in the class of regular languages, and gave languages meeting these bounds. In 1994 these operations, along with intersection, reversal, and left and right quotients, were studied in detail by Yu, Zhuang and K. Salomaa [42].

State complexity of operations has also been studied in several proper subclasses of regular languages. Surprisingly, in the class of star-free languages studied by Brzozowski and Liu [10], the operations union, intersection, difference, symmetric difference, concatenation and star meet the bounds for arbitrary regular languages; in the case of reversal, the bound 2^n cannot be reached [11], but $2^n - 1$ is attainable. In general, however, the bounds are quite different in different classes. In addition to the star-free class, the following classes have been considered: unary languages (Yu, Zhuang and K. Salomaa [42], Pighizzini and Shallit [35]); finite languages (Câmpeanu, Culik, K. Salomaa and Yu [12], Yu [41], Han and K. Salomaa [16]); cofinite languages (Bassino, Giambruno and Nicaud [2]); right-, left-, two-sided and all-sided ideals (Brzozowski, Jirásková and Li [7]); prefix-, suffix-, factor- and subword-closed languages (Brzozowski, Jirásková and Zou [9]); union-free languages (Jirásková and Masopust [22], Jirásková and Nagy [24]); non-returning languages (Eom, Han and Jirásková [14]); reversal in $\mathcal{R}$-trivial and $\mathcal{J}$-trivial languages (Jirásková and Masopust [23]); and operations on prefix- and suffix-free languages discussed below in more detail.

Languages that are prefix-, suffix-, bifix-, factor- (also called infix-), and subword-free will be called *xfix-free*. Xfix-free languages (with the exception of $\{\varepsilon\}$, where ε is the empty word) are codes, which constitute an important class of languages and have applications in such areas as cryptography, data compression, and information transmission. They have been studied extensively; see, for example, [3, 27]. In particular, *prefix* and *suffix codes* [3] are prefix-free and suffix-free languages, respectively, *infix codes* [36, 37] are factor-free, and *hypercodes* [36, 37] are subword-free, where by subword we mean subsequence. Moreover, xfix-free languages are special cases of convex[1] languages [1, 38]. We are interested only in regular xfix-free languages.

The state complexities of intersection, union, concatenation, star, and reversal for prefix-free languages were first studied by Han, K. Salomaa and Wood [18]. These results have been strengthened by Jirásková and Krausová in [21] who provided witnesses over smaller alphabets. The same operations for suffix-free languages were studied by Han and K. Salomaa [17]. The upper bounds for suffix-free languages from [17] have been shown to be tight in the binary case for union and intersection by Jirásková and Olejár [25], and for star by Cmorik [13]. On the other hand, as shown in [13], the upper bound for reversal of suffix-free languages cannot be met in the binary case. In [13, 20, 21, 30], some

[1] A language is prefix-convex if it satisfies the condition that, if a word w and its prefix u are in the language, then so is every prefix of w that has u as a prefix. In a similar way, we define suffix-, bifix-, factor-, and subword-convex languages.

other operations on prefix- and suffix-free languages, such as difference, symmetric difference, left quotient, and cyclic shift have been studied, and tight bounds with witnesses over optimal alphabets have been found.

In this paper, we study bifix-, factor- and subword-free languages. In particular, we obtain tight upper bounds on the complexities of intersection, union, difference, symmetric difference, star, product (concatenation), and reversal in these three classes of xfix-free languages.

A much shorter version of this paper appeared in [8]. In the present paper we have added several new results on binary bifix- and factor-free languages.

2 Preliminaries

It is assumed that the reader is familiar with finite automata and regular languages as treated in [34, 40], for example. If Σ is a finite non-empty *alphabet*, then Σ^* is the set of all words over this alphabet, with ε as the *empty word*. For $w \in \Sigma^*$, let $|w|$ be the length of w. A *language* is any subset of Σ^*.

The following set operations are defined on languages: *complement* ($\overline{L} = \Sigma^* \setminus L$), *union* ($K \cup L$), *intersection* ($K \cap L$), *difference* ($K \setminus L$), and *symmetric difference* ($K \oplus L$). All four of these Boolean operation with two arguments are denoted by $K \circ L$.

We also define the *product,* usually called *concatenation* or *catenation,* ($KL = \{w \in \Sigma^* \mid w = uv, u \in K, v \in L\}$), (Kleene) *star* ($L^* = \bigcup_{i \geq 0} L^i$ with $L^0 = \{\varepsilon\}$), and *positive closure* ($L^+ = \bigcup_{i \geq 1} L^i$).

The *reverse* w^R of a word $w \in \Sigma^*$ is defined inductively as follows: $\varepsilon^R = \varepsilon$, and $(wa)^R = aw^R$ for every symbol a in Σ and every word w in Σ^*. The *reverse* of a language L is denoted by L^R and is defined as $L^R = \{w^R \mid w \in L\}$.

Regular languages over Σ are languages that can be obtained from the *set of basic languages* $\{\varnothing, \{\varepsilon\}\} \cup \{\{a\} \mid a \in \Sigma\}$, using a finite number of operations of union, product, and star. We use regular expressions to represent languages. If E is a regular expression, then $\mathcal{L}(E)$ is the language denoted by that expression. For example, the regular expression $E = (\varepsilon \cup a)^* b$ denotes language $L = \mathcal{L}(E) = (\{\varepsilon\} \cup \{a\})^* \{b\}$. We usually do not distinguish notationally between regular languages and regular expressions.

The *ε-function* L^ε of a regular language L is $L^\varepsilon = \varnothing$ if $\varepsilon \notin L$, and $L^\varepsilon = \{\varepsilon\}$ if $\varepsilon \in L$. We usually write the language $\{\varepsilon\}$ as ε. Then the ε-function can be computed inductively as follows:

$$a^\varepsilon = \begin{cases} \varnothing, & \text{if } a = \varnothing, \text{ or } a \in \Sigma; \\ \varepsilon, & \text{if } a = \varepsilon. \end{cases} \tag{1}$$

$$(\overline{L})^\varepsilon = \begin{cases} \varnothing, & \text{if } L^\varepsilon = \varepsilon; \\ \varepsilon, & \text{if } L^\varepsilon = \varnothing. \end{cases} \tag{2}$$

$$(K \cup L)^\varepsilon = K^\varepsilon \cup L^\varepsilon, \quad (KL)^\varepsilon = K^\varepsilon \cap L^\varepsilon, \quad (L^*)^\varepsilon = \varepsilon. \tag{3}$$

The *quotient* [4] of a language L by a word w is defined as $L_w = \{x \in \Sigma^* \mid wx \in L\}$. Quotients of regular languages [4, 5] can be computed as follows. The *quotient by a letter*

a in Σ is computed by induction:

$$b_a = \begin{cases} \varnothing, & \text{if } b \in \{\varnothing, \varepsilon\}, \text{ or } b \in \Sigma \text{ and } b \neq a; \\ \varepsilon, & \text{if } b = a. \end{cases} \tag{4}$$

$$(\overline{L})_a = \overline{L_a}, \quad (K \cup L)_a = K_a \cup L_a, \quad (KL)_a = K_a L \cup K^\varepsilon L_a, \quad (L^*)_a = L_a L^*. \tag{5}$$

The quotient by a word w in Σ^* is computed by induction on the length of w:

$$L_\varepsilon = L, \quad L_{wa} = (L_w)_a. \tag{6}$$

By convention, L_w^ε always means $(L_w)^\varepsilon$.

A *deterministic finite automaton* (dfa) is a quintuple $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite non-empty set of *states*, Σ is a finite *alphabet*, $\delta\colon Q \times \Sigma \to Q$ is the *transition function*, q_0 is the *initial state*, and $F \subseteq Q$ is the set of *final* states. As usual, the transition function is extended to $Q \times \Sigma^*$. Dfa $\mathcal{D}$ accepts a word w in Σ^* if $\delta(q_0, w) \in F$. The set of all words accepted by $\mathcal{D}$ is $L(\mathcal{D})$, the language accepted by $\mathcal{D}$. By the *language of a state* q of $\mathcal{D}$ we mean the language L_q accepted by the automaton $(Q, \Sigma, \delta, q, F)$. Two states of $\mathcal{D}$ are *equivalent* if their languages are equal. A state is *empty* if its language is empty.

A quotient L_w is *final* if $\varepsilon \in L_w$; otherwise it is *non-final*. The *quotient automaton* of a regular language L is the automaton in which the quotients of the language are states; it is formally defined as the dfa $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{L_w \mid w \in \Sigma^*\}$, $\delta(L_w, a) = L_{wa}$, $q_0 = L_\varepsilon$, $F = \{L_w \mid \varepsilon \in L_w\}$. This is a minimal dfa accepting L. The number of distinct quotients of a language is called its *quotient complexity* and is denoted by $\kappa(L)$. Hence the quotient complexity of L is equal to the state complexity of L, and we call it simply *complexity*. Whenever convenient, we derive upper bounds on the quotient complexity of operations on xfix-free languages following the approach of [5].

A *nondeterministic finite automaton* (nfa) is a quintuple $\mathcal{N} = (Q, \Sigma, \delta, I, F)$, where Q, Σ, and F are as in a dfa, $\delta\colon Q \times \Sigma \to 2^Q$ is the nondeterministic transition function, and I is the set of initial states. We extend the transition function to a function $\delta\colon 2^Q \times \Sigma^* \to 2^Q$ as usual. The language accepted by $\mathcal{N}$ is $L(\mathcal{N}) = \{w \in \Sigma^* \mid \delta(I, w) \cap F \neq \varnothing\}$. Two nfas are *equivalent* if their languages are equal.

A *reverse* of a dfa $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ is an nfa $\mathcal{D}^R = (Q, \Sigma, \delta^R, F, \{q_0\})$, where $\delta^R(q, a) = \{p \in Q \mid \delta(p, a) = q\}$. The nfa $\mathcal{D}^R$ accepts the language $(L(\mathcal{D}))^R$.

Every nondeterministic finite automaton $\mathcal{N} = (Q, \Sigma, \delta, I, F)$ can be converted to an equivalent dfa $\mathcal{D} = (2^Q, \Sigma, \delta', I, F')$, where $F' = \{R \in 2^Q \mid R \cap F \neq \varnothing\}$ and $\delta'(R, a) = \cup_{r \in R} \delta(r, a)$ for each R in 2^Q and each a in Σ. We call this dfa $\mathcal{D}$ the *subset automaton* of nfa $\mathcal{N}$. The subset automaton need not be minimal since some of its states may be unreachable or equivalent.

3 Xfix-Free Languages

If $u, v, w, x \in \Sigma^*$ and $w = uxv$, then u is a *prefix* of w, x is a *factor* of w, and v is a *suffix* of w. Both u and v are also factors of w. If $w = u_0 v_1 u_1 \cdots v_n u_n$, where $u_i, v_i \in \Sigma^*$, then $v = v_1 v_2 \cdots v_n$ is a *subword* of w. Every factor of w is also a subword of w.

A language L is *prefix-free* (respectively, *suffix-*, *factor-*, or *subword-free*) if, whenever words u and v are in L and u is a prefix (respectively, suffix, factor, or subword) of v, then $u = v$. Additionally, L is *bifix-free* if it is both prefix- and suffix-free. All subword-free languages are factor-free, and all factor-free languages are bifix-free.

If ε is a quotient of L, then L also has the empty quotient, since $\varepsilon_a = \varnothing$, for all a in Σ. We say that a quotient L_w is *uniquely reachable* if $L_w = L_x$ implies that $w = x$. We now restate two results from [17, 18] in our terminology.

Remark 1. A non-empty language is prefix-free if and only if it has exactly one final quotient and that quotient is ε.

Remark 2. If L is a non-empty suffix-free language, then it has the empty quotient and $L_\varepsilon = L$ is uniquely reachable.

Let L be any language. If $(L_u)_x = L_v$ for some words u, v and a non-empty word x, then L_v is *positively reachable* from L_u, and we denote this by $L_u \to L_v$. The relation $\to$ is transitive. The next remark uses this relation to characterize finite languages.

Remark 3. If L is any language with $\{L_1, L_2, \ldots, L_n\}$ as its set of quotients, and u and v are words in Σ^*, then the following are equivalent:

1. L is finite.
2. $L_u \to L_v$ and $L_v \to L_u$ if and only if $L_u = L_v = \varnothing$.
3. There exists a total order $L = L_1 \prec L_2 \prec \cdots \prec L_{n-1} \prec L_n = \varnothing$, on the set of quotients such that for any $x \in \Sigma^+$, $(L_i)_x = L_j$ implies $L_i \prec L_j$ or $L_i = L_j = L_n$.

Note that every subword-free language is finite [15]. The next lemma will be used later to prove that upper bounds on the quotient complexity of some operations on subword-free languages cannot be reached if the alphabet of the language does not have sufficiently many letters.

Remark 4. Let L be a finite language with $\kappa(L)$, where $n \geq 4$. Let the distinct quotients $L = L_\varepsilon = L_1, L_2, \ldots, L_{n-2}, L_{n-1} = \varepsilon, L_n = \varnothing$ of L be ordered as in Remark 3. If $L_w = L_2$ for some word w, then $|w| = 1$.

Finally, we describe a simple method of constructing xfix-free languages.

Proposition 1. *Let $L \subseteq \Sigma^*$ be any language, and let $a \notin \Sigma$. Then (1) aL is suffix-free, (2) La is prefix-free, (3) aLa is factor-free.*

4 Boolean Operations

For prefix-free languages, it was shown in [18, 21] that the tight bounds for union, intersection, difference, and symmetric difference are $mn-2$, $mn-2(m+n-3)$, $mn-(m+2n-4)$, and $mn-2$, respectively. For union and intersection of suffix-free languages, the tight bounds are $mn-(m+n-2)$ and $mn-2(m+n-3)$, respectively [17]. The bounds

for difference and symmetric difference are $mn-(m+2n-4)$ and $mn-(m+n-2)$, respectively [25], and the bounds for all four Boolean operations are met by binary suffix-free languages [13]. The next theorem provides results for Boolean operations on bifix- and factor-free languages.

Theorem 1 (Boolean Operations: Bifix- and Factor-Free Languages). *Let K and L be bifix-free languages over an alphabet Σ with $\kappa(K)=m$ and $\kappa(L)$, where $m,n\geq 4$. Then*

1. $\kappa(K\cap L)\leq mn-3(m+n-4)$;
2. $\kappa(K\setminus L)\leq mn-(2m+3n-9)$;
3. $\kappa(K\cup L),\kappa(K\oplus L)\leq mn-(m+n)$.

The bounds for intersection and difference can be met by binary factor-free languages, and the bound for union and symmetric difference can be met by ternary factor-free languages.

Proof. We first derive the upper bound for bifix-free languages; this bound applies also to factor-free languages. Since $(K\circ L)_w=K_w\circ L_w$, the bound for regular languages is mn; however, not all pairs (K_i,L_j) of quotients of K and L can occur if the languages are xfix-free.

If K and L are bifix-free, by unique reachability we get a reduction of $m-1+n-1=m+n-2$ from the general bound mn, because pairs of the form (K_ε,L_w) and (K_w,L_ε) can occur only if $w=\varepsilon$.

Moreover, both languages K and L have ε and $\varnothing$ as quotients. For intersection, we have $\varnothing\cap L_w=K_w\cap\varnothing=\varnothing$, and this results in another reduction of $m-2+n-2$ quotients. Also, the quotients $\varepsilon\cap L_w$ and $K_w\cap\varepsilon$ are either empty or equal to ε; this gives an additional reduction of $m-3+n-3$ states. Altogether, we get the upper bound.

For difference, we eliminate $m+n-2$ quotients by unique reachability, $n-2$ quotients by the fact that $\varnothing\setminus L_w=\varnothing$ (keeping only one representative $\varnothing\setminus\varnothing$), $m-2$ quotients by the fact that $K_w\setminus\varnothing=K_w\setminus\varepsilon$ (keeping $K_w\setminus\varnothing$ as a representative), and $n-3$ more quotients by the rule $\varepsilon\setminus L_w=\varepsilon$, for a total reduction of $2m+3n-9$. For union we have the unique reachability reduction of $m+n-2$, and a further reduction of 2 by the rule $\varepsilon\cup\varepsilon=\varepsilon\cup\varnothing=\varnothing\cup\varepsilon=\varepsilon$. For symmetric difference we have the unique reachability reduction of $m+n-2$, and a further reduction of 2 in view of the fact that $\varepsilon\oplus\varepsilon=\varnothing\oplus\varnothing=\varnothing$ and $\varepsilon\oplus\varnothing=\varnothing\oplus\varepsilon=\varepsilon$.

For the tightness of the bounds for intersection and difference, let K and L be the binary factor-free languages accepted by the quotient automata of Figure 1, where missing transitions in the automaton accepting K (L) all go to the empty state m (n). In the corresponding cross-product automaton of Figure 2, no states in row 1 or column 1 are reachable, except for $(1,1)$. Also, states $(m-1,2)$ and $(m,2)$ are unreachable, as are the states in column $n-1$, except $(3,n-1)$, $(m-1,n-1)$, and $(m,n-1)$. The remaining states are all reachable.

For intersection, the only final state is $(m-1,n-1)$, and all the other states in the last two rows and columns are empty. We will prove that states $(1,1)$, (i,j) with $2\leq i\leq m-2$

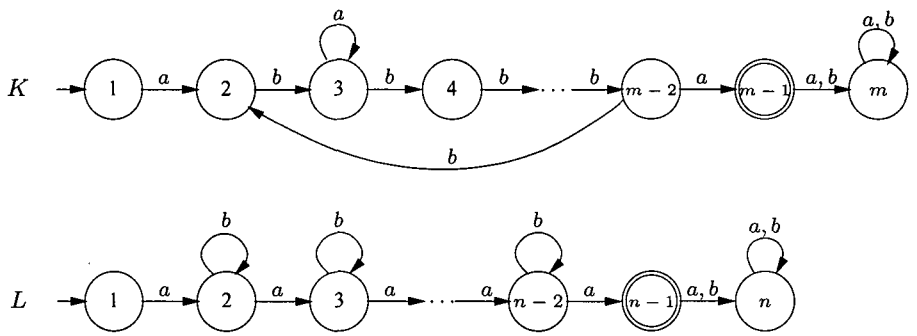


Figure 1: Binary factor-free witnesses for intersection and difference.

and $2 \leq j \leq n-2$, $(m-1, n-1)$, and (m, n) (which represents all the empty states) are all distinguishable. Then it follows that $\kappa(K \cap L) \geq (m-3)(n-3)+3 = mn-3(m+n-4)$.

State (m, n) is the only empty state in our set. We show that for each other non-final state (i, j), there exists a word w_{ij} that is accepted only from state (i, j). We have $w_{m-2,n-2} = a$ because word a is accepted only from state $(m-2, n-2)$. Since only one transition on letter b goes to state $(m-2, n-2)$, and it goes from state $(m-3, n-2)$, the word ba is accepted only from state $(m-3, n-2)$. Therefore $w_{m-3,n-2} = ba = bw_{m-2,n-2}$. For similar reasons we have

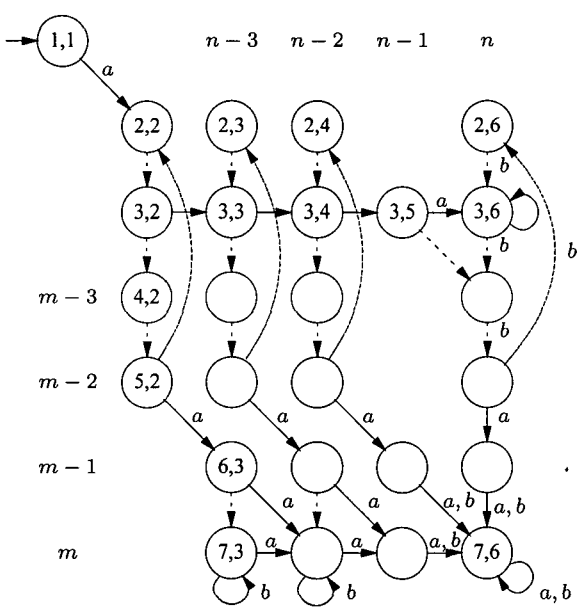


Figure 2: Cross-product automaton for $m = 6, n = 7$. Missing transitions go to $(7, 6)$.

$w_{i,n-2} = bw_{i+1,n-2}$ for $i = 2, 3, \ldots, m-3$,
$w_{3j} = aw_{3,j+1}$ for $j = 2, 3, \ldots, n-3$,
$w_{2j} = bw_{3j}$ for $j = 2, 3, \ldots, n-3$,
$w_{m-2,j} = bw_{2j}$ for $j = 2, 3, \ldots, n-3$,
$w_{ij} = bw_{i+1,j}$ for $i = 4, 5, \ldots, m-3$ and $j = 2, 3, \ldots, n-3$,
$w_{11} = aw_{22}$,

which proves that $mn - 3(m+n-4)$ states are pairwise distinguishable.

In the case of difference, all the states in row m, as well as state $(m-1, n-1)$ are empty. All the other states in row $m-1$ accept ε, and so are equivalent. For each i with $2 \leq i \leq m-2$, states $(i, n-1)$ and (i, n) are equivalent. Among the other reachable states consider two distinct states p and q. If they are in different rows, then by a word in b^* we can send p to a state p' in row 3, and q to a state q' that is not in row 3. Now by a^n, state q' goes to the empty state, while p' goes to state $(3, n)$ that is not empty. Two distinct states in the same row go by a word in b^* to row 3. Then, by a word in a^*, the first goes to state $(3, n-2)$ while the second to $(3, n)$, and now $b^{m-2-3}a$ distinguishes them. In summary, $\kappa(K \setminus L) \geq (m-3)(n-3) + m - 3 + 3 = mn - (2m + 3n - 9)$.

To prove the tightness of the bounds for union and symmetric difference, consider the languages $K = a(c^*(a \cup b))^{m-3}$, $L = a(b^*(a \cup c))^{n-3}$; see Figure 3, where missing transitions in the automaton accepting K (L) all go to the empty state m (n). If $w \in K$, then $w = av$ for some word v containing $m-3$ occurences of symbols from $\{a, b\}$ and ending in a or b. Thus no proper factor of w is in K, and so K is factor-free. A similar proof applies to L.

In the cross-product automaton of Figure 4 for the Boolean operations on languages K and L, all the states are reached from the initial state $(1, 1)$ by a word in $ab^*c^* \cup ac^*b^*$, except for state $(m-1, n-1)$ which is reached from state $(m-2, n-2)$ by a.

For union, all the states in row $m-1$ and in column $n-1$ are final, and moreover, the three states $(m, n-1)$, $(m-1, n-1)$, and $(m-1, n)$ are equivalent. The word ab^{m-3} is accepted only from $(1, 1)$. Consider two distinct non-final states (i, j) and (k, ℓ). If $i < k$, then $c^n b^{m-1-i}$ is accepted from (i, j) but not from (k, ℓ). If $j < \ell$, then $b^m c^{n-1-j}$ is accepted from (i, j) but not from (k, ℓ). Now consider two distinct final states different

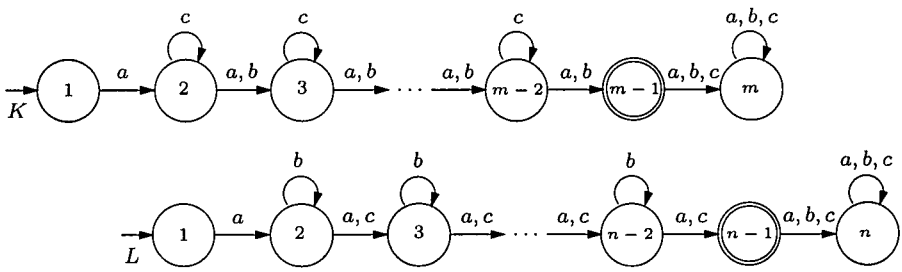


Figure 3: Ternary factor-free languages witnesses for union and symmetric difference.

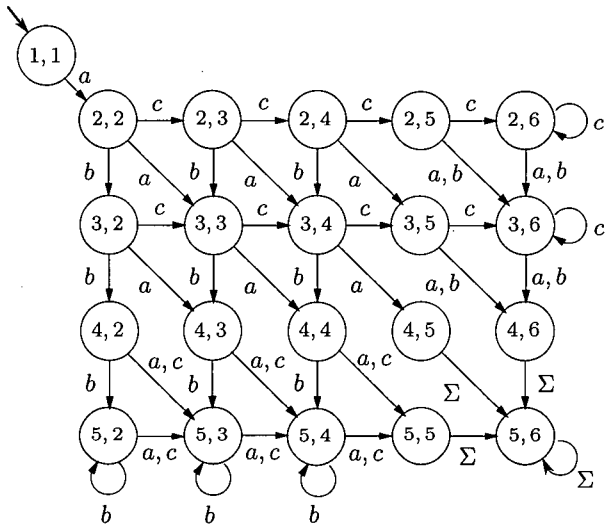


Figure 4: Cross-product automaton for Boolean operations on languages from Figure 3.

from $(m, n-1)$ and $(m-1, n)$. By c, the two states either go to two states one of which is final and the other non-final, or to two distinct non-final, and hence distinguishable, states. This proves distinguishability of $mn - (m+n)$ states.

The proof for symmetric difference is the same as for union, except that now state $(m-1, n-1)$ is empty and states $(m, n-1)$ and $(m-1, n)$ are equivalent. □

The next proposition gives lower bounds for union and symmetric difference of binary bifix-free languages.

Proposition 2 (Union, Symmetric Difference: Binary Bifix-Free Languages; Lower Bound). *Let $m, n \geq 6$. There exist binary bifix-free languages K and L with $\kappa(K) = m$ and $\kappa(L) = n$ such that $\kappa(K \cup L), \kappa(K \oplus L) \geq mn - (m+n) - 2$.*

Proof. Consider the binary languages

$$\begin{aligned} K &= a((ba^*)^{m-5}b \cup a)(b((ba^*)^{m-5}b \cup a))^*a, \\ L &= a(a \cup b)^{n-4}(b(a \cup b)^{n-4})^*a. \end{aligned}$$

Quotient automata for $m = 7$ and $n = 6$ are shown in Figure 5. Since both languages have ε as the only final quotient, they are prefix-free. Since the reverse automata are deterministic, the reversed languages also have ε as the only final quotient, and so are prefix-free. Thus both languages are bifix-free.

The cross-product automaton is shown in Figure 6. States in row 1 and column 1 are unreachable, with the exception of the initial state (1,1). Also, states $(2, n-1)$ and

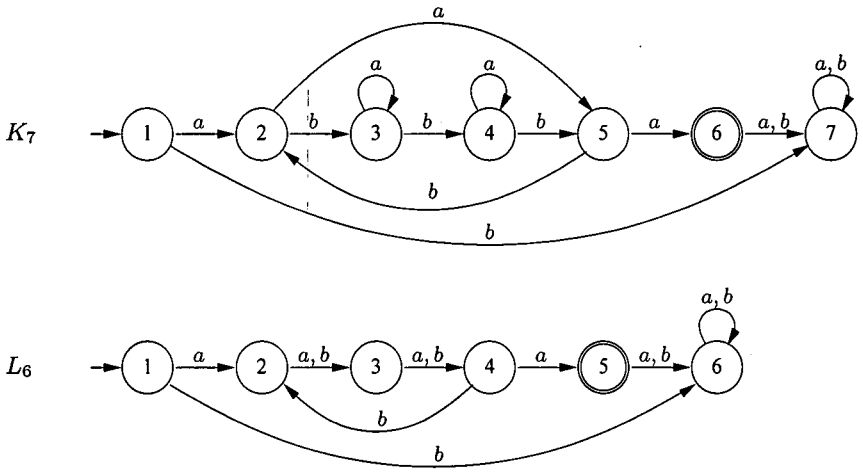


Figure 5: Binary bifix-free languages meeting the bound $mn - (m+n) - 2$ for union and symmetric difference.

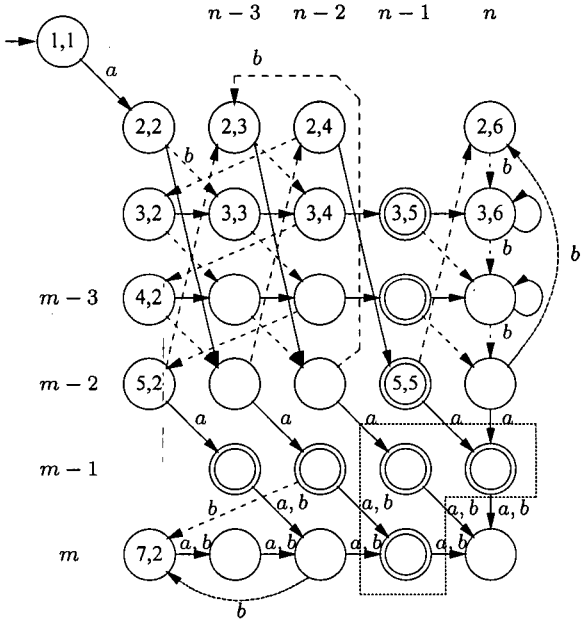


Figure 6: Cross-product automaton for automata from Figure 5, where dashed-transitions are on input b, and missing transitions go to state (7,6).

$(m-1,2)$ are unreachable. The initial state $(1,1)$ goes to state $(2,2)$ by a and then to state $(3,3)$ by b. From $(3,3)$, all the other states in row 3, except for $(3,2)$ are reached by a-transitions. Next, state $(3,n-2)$ goes to state $(4,2)$ by b, and then to $(4,j)$ by a^{j-2} $(3 \leq j \leq n)$. In this way, all the states in rows $4,5,\ldots,m-3$ can be reached. State $(m-3,n-2)$ goes to state $(m-2,2)$ by b, and states $(m-2,j)$ with $j \geq 3$, except for state $(m-2,n-1)$ that is reached from $(2,n-2)$ by a, are reached from states $(m-3,j-1)$ by b. States $(2,j)$ with $j \geq 3$, except for $(2,n-1)$, are reached from $(m-2,j-1)$ by b. State $(2,n-2)$ goes to $(3,2)$ by b. From states in row $m-2$ all reachable states in row $m-1$ are reached by a. State $(m,2)$ is reached by b from $(m-1,n-2)$; from here, all the other states is row m are reached by words in a^*.

For union, the three final states $(m-1,n-1)$, $(m-1,n)$ and $(m,n-1)$ are equivalent. Consider the other reachable states. First, let $p=(i,j)$ and $q=(k,\ell)$ be two non-final states with $i<k$. We can use b-transitions to get p into a state p' in row 3, and q into a state q' in a row i with $i \neq 3$. By a^n, state p' goes to $(3,n)$, while q' goes to (i,n). Now $b^{m-2-3}a$ is accepted from $(3,n)$ but not from (i,n). Next, let p and q be two distinct non-final states in the same row. If they are in the last row, then a word in a^* distinguishes them. Otherwise, we can get them into states $(3,j)$ and $(3,\ell)$ with $j<\ell$, using b-transitions. Now $(3,j)$ accepts a^{n-1-j} while $(3,\ell)$ goes to the non-final state $(3,n)$. Finally, consider two distinct final states different from $(m-1,n)$, $(m,n-1)$. By b, they go to two distinct non-final, and so distinguishable, states. The proof for symmetric difference is similar, except that now state $(m-1,n-1)$ is empty. □

We now show that the upper bound for union and symmetric difference of binary bifix-free languages is the same as the lower bound in the proposition above.

Proposition 3 (Union, Symmetric Difference: Binary Bifix-Free Languages; Upper Bound). *Let $m,n \geq 4$ and let K and L be binary bifix-free languages with $\kappa(K)=m$ and $\kappa(L)$. Then $\kappa(K \cup L), \kappa(K \oplus L) \leq mn-(m+n)-2$.*

Proof. Let K be a bifix-free language accepted by the quotient automaton $\mathcal{A}$ over $\{a,b\}$ with states $1,2,\ldots,m$, where 1 is the initial state, $m-1$ is the only final state and it accepts only ε, and m is the empty state. Let L be a similar language accepted by $\mathcal{B}$ with states $1,2,\ldots,n$, initial state 1, final state $n-1$ accepting ε, and empty state n.

Construct the corresponding cross-product automaton with states (i,j), where i is a state of $\mathcal{A}$ and j is a state of $\mathcal{B}$. In this cross-product automaton, we cannot go from rows $m-1$ and m to any state (i,j) with $i<m-1$, and similarly, we cannot go from columns $n-1$ and n to and state (i,j) with $j<n-1$.

If state 1 of $\mathcal{A}$ goes by both inputs a and b to a state in $\{m-1,m\}$, then no row i with $i<m-1$ can be reached. Therefore, if the bound is to be met, at least one input, say a, takes state 1 to a state i with $i<m-1$. Suppose also that b takes 1 to a state in $\{m-1,m\}$. A similar condition applies to L. Suppose that input b takes state 1 of $\mathcal{B}$ to a state j with $j<n-1$, and a, to a state in $\{n-1,n\}$. Then no state (i,j) with $i<m-1$ and $j<n-1$ can be reached. It follows that, without loss of generality, each automaton must take its initial state by a to a state that is neither final nor empty; for convenience, let this state be 2 in both automata. Then no other transition by a may go to state 2 in the two automata, otherwise they would not be suffix-free.

It follows that in the cross-product automaton, all the states in row 2 and column 2, except for $(2,2)$, must be reached from some states by input b. Thus, if all the states are reachable, there must be an incoming transition by b to each state i with $i \geq 2$ in $\mathcal{A}$ and j with $j \geq 2$ in $\mathcal{B}$. In particular, if state $(m-1,2)$ or $(2,n-1)$ is reachable, then some state, say p_1 (respectively q_1) different from $m-1$ (respectively $n-1$) must go to state $m-1$ (respectively $n-1$) in $\mathcal{A}$ (respectively $\mathcal{B}$). Now since p_1 goes to $m-1$ by b, it cannot go anywhere else by b. Thus there must be some other state p_2 not in $\{p_1, m-1, m\}$ that goes to p_1 by b. Then there must be a state p_3 not in $\{p_2, p_1, m-1, m\}$ that goes to p_2 by b, and so on. Eventually, we have $p_{m-3} \xrightarrow{b} p_{m-4} \xrightarrow{b} \cdots \xrightarrow{b} p_3 \xrightarrow{b} p_2 \xrightarrow{b} p_1 \xrightarrow{b} m-1 \xrightarrow{b} m$, where all the states are pairwise distinct, and no state, except possibly state 1, goes by b to state p_{m-3}.

First assume state 1 goes to state p_{m-3} by b. If $p_{m-3} = 2$, then 1 goes to 2 by a and by b. This means that there is no other transition to state 2, and so row 2 is not reachable in the cross-product automaton. If $p_{m-3} > 2$ and 1 goes to p_{m-3} by b, then no other state goes to p_{m-3} by b because of suffix-freeness, and so row p_{m-3} may only be reached by a's. However, in such a case $(p_{m-3}, 2)$ is unreachable, since it is in row p_{m-3} that can be reached only by a's and at the same time in column 2 that can be reached only by b's.

Now assume that there is no transition by b going to state p_{m-3}. If $p_{m-3} \geq 3$, then $(p_{m-3}, 2)$ is unreachable. If $p_{m-3} = 2$, then the whole row 2, except for $(2,2)$ is unreachable. The same considerations hold for automaton $\mathcal{B}$. This gives the desired upper bound $mn - (m+n) - 2$. □

As a corollary of the two propositions above, we get the tight bound on the complexity of union and symmetric difference of binary bifix-free languages.

Theorem 2 (Union, Symmetric Difference: Binary Bifix-Free Languages). *Let K and L be binary bifix-free languages with $\kappa(K) = m$ and $\kappa(L)$, where $m, n \geq 6$. Then $\kappa(K \cup L), \kappa(K \oplus L) \leq mn - (m+n) - 2$, and the bound is tight.*

In a recent paper [19] Iván has shown that $f(m,n) = mn-(m+n)-2-\lfloor \frac{\min\{m,n\}-2}{2} \rfloor$ is a lower bound on the union of binary factor-free languages, and that $f(m,n) - 1$ is a lower bound for symmetric difference.

We now turn our attention to subword-free languages. The next theorem gives tight bounds for all four Boolean operations and shows that the bounds cannot be met using a fixed alphabet.

Theorem 3 (Boolean Operations: Subword-Free Languages). *Suppose that K and L are subword-free languages over an alphabet Σ with $\kappa(K) = m$ and $\kappa(L)$, where $m, n \geq 4$. Then*

1. *$\kappa(K \cup L), \kappa(K \oplus L) \leq mn - (m+n)$, and the bound is tight if $|\Sigma| \geq m+n-3$;*
2. *$\kappa(K \cap L) \leq mn - 3(m+n-4)$, and the bound is tight if $|\Sigma| \geq m+n-7$;*
3. *$\kappa(K \setminus L) \leq mn - (2m+3n-9)$, and the bound is tight if $|\Sigma| \geq m+n-6$.*

Moreover, the bounds cannot be met for smaller alphabets.

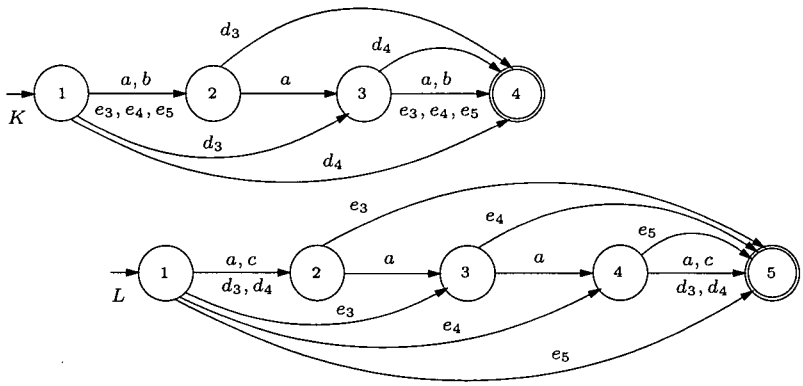


Figure 7: Subword-free witness languages for Boolean operations; $m = 5$, $n = 6$.

Proof. Since subword-free languages are bifix-free, all the upper bounds apply. To prove tightness, let $\Sigma = \{a, b, c\} \cup \{d_i \mid 3 \leq i \leq m-1\} \cup \{e_j \mid 3 \leq j \leq n-1\}$. Consider the languages K and L defined by the following quotient equations:

$$\begin{aligned}
K_1 &= (a \cup b \cup e_3 \cup \cdots \cup e_{n-1})K_2 \cup \textstyle\bigcup_{i=3}^{m-1} d_i K_i, \\
K_i &= aK_{i+1} \cup d_{i+1}K_{m-1} \qquad i = 2, 3, \ldots, m-3, \\
K_{m-2} &= (a \cup b \cup d_{m-1} \cup e_3 \cup e_4 \cup \cdots \cup e_{n-1})K_{m-1}, \\
K_{m-1} &= \varepsilon, \\
K_m &= \varnothing,
\end{aligned}$$

$$\begin{aligned}
L_1 &= (a \cup c \cup d_3 \cup \cdots \cup d_{m-1})L_2 \cup \textstyle\bigcup_{j=3}^{n-1} e_j L_j, \\
L_j &= aL_{j+1} \cup e_{j+1}L_{n-1} \qquad j = 2, 3, \ldots, n-3, \\
L_{n-2} &= (a \cup c \cup e_{n-1} \cup d_3 \cup d_4 \cup \cdots \cup d_{m-1})L_{n-1}, \\
L_{n-1} &= \varepsilon, \\
L_n &= \varnothing.
\end{aligned}$$

The dfa's (minus empty states) for languages K and L, where $m = 5$ and $n = 6$, are shown in Figure 7. We now show that languages K and L are subword-free. For this purpose, let

$$\Gamma = \{a, b, e_3, e_4, \ldots, e_{n-1}\}, \text{ and } \Delta = \{d_3, d_4, \ldots, d_{m-1}\}.$$

Notice that no word in Γ^* of length less than $m-2$ is in K. Now let w be a word in language K. Then word w either contains no letter from Δ, or contains at most two such letters. If w contains no letter from Δ, then w is a word in Γ^* of length $m-2$, and so no its proper subword is in K. If w contains exactly one letter from Δ, then either $w = ud_i$ for some word u in Γ^* of length $i-2$, or $w = d_i v$ for some word v in Γ^* of length $m-1-i$. In both cases, no proper subword of w is in language K. Finally, if w contains two letters from Δ, then $w = d_i a^k d_{i+k+1}$ where $k \geq 0$ and $3 \leq i < i+k+1 \leq m-2$. No proper subword of such a word is in language K. This means that language K is subword-free. The proof for language L is similar.

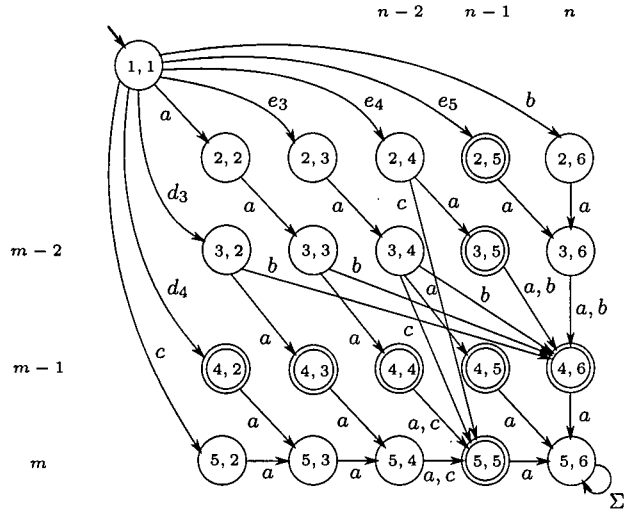


Figure 8: Reachability in the cross-product automaton for the union of languages from Figure 7 and transitions by b and c.

Figure 8 depicts the cross-product automaton of dfa's from Figure 7, where we show only the transitions necessary to prove reachability and those caused by b and c. The states in the first row and the first column, except for the initial state $(1,1)$, are unreachable. Now consider the remaining states. All the states in the second row and the second column are reached from $(1,1)$ by symbols in Σ. Each other state is reached from a state in the second row or second column by a word in a^*.

For union, all the states in row $m-1$ and in column $n-1$ are final, and the three states $(m, n-1)$, $(m-1, n-1)$, and $(m-1, n)$ accept only ε, and so are equivalent. These three states are distinguishable from all other final states, since each of the other final states accepts at least one non-empty word. Now let (i,j) and (k,ℓ) be two distinct states other than the three states accepting only word ε. First assume that $i < k$. If $i = m-1$, then state (i,j) is final while state (k,ℓ) is non-final. If $i \leq m-2$, then $a^{m-2-i}b$ is accepted from state (i,j), but not from state (k,ℓ). Symmetrically, if $j < \ell$, then either ε or $a^{n-2-j}c$ distinguishes the two states. Therefore all the $mn-(m+n)$ states are pairwise distinguishable. For symmetric difference, $(m-1, n-1)$ is empty; the rest of the proof is the same as for union.

For intersection, the only final state is $(m-1, n-1)$, and all the non-final states in the last two rows and last two columns are empty. Next, the word a is accepted only from state $(m-2, n-2)$, the word d_i $(3 \leq i \leq m-2)$ is accepted only from state $(i-1, n-2)$, while the word e_j $(3 \leq i \leq n-2)$, only from state $(m-2, j-1)$. This means that for each state (i,j), there exists a word in $a^*(a \cup d_3 \cup \cdots \cup d_{m-2} \cup e_3 \cup \cdots \cup e_{n-2})$ that is accepted only from (i,j). So we get $mn - 3(m+n-4)$ pairwise distinguishable states. Notice, that here we do not use transitions by symbols b, c, d_{m-1}, e_{n-1}, and so we can

simply omit these symbols to get witness languages over an alphabet of size $m + n - 7$.

For difference, all the states in row $m-1$, except for state $(m-1, n-1)$, are final and accept ε. All the states in the last row, as well as state $(m-1, n-1)$, are empty, and states $(i, n-1)$ and (i, n) with $2 \leq i \leq m-2$ are equivalent. States in different rows (up to row $m-1$) are distinguished by a word in a^*b. States in row $m-2$ are distinguished by a word in $a \cup e_3 \cup e_4 \cup \cdots \cup e_{n-2}$ because a distinguishes states $(m-2, n-2)$ and $(m-2, n-1)$, and if $2 \leq j < \ell \leq n-1$ and $j \neq n-2$, then word e_{j+1} is not accepted from $(m-2, j)$ but is accepted from $(m-2, \ell)$. Next, states $(i, n-2)$ and $(i, n-1)$ with $2 \leq i \leq m-3$ are distinguished by d_{i+1}. Finally, if two distinct states are in the same row, then there is a word in a^*, by which the two states either go to two distinct states in row $m-2$, or to two states $(i, n-2)$ and $(i, n-1)$ with $2 \leq i \leq m-3$. In both cases the resulting states are distinguishable, which proves the distinguishability of $mn - (2m + 3n - 9)$ states. Notice that now we do not use transitions by c, d_{m-1}, e_{n-1}, and so the bound is met for an alphabet of size $m + n - 6$.

We now show that the upper bounds cannot be met using smaller alphabets. Let the quotients of K and L be $K = K_1, K_2, \ldots, K_{m-2}, K_{m-1} = \varepsilon, K_m = \varnothing$, and $L = L_\varepsilon = L_1, L_2, \ldots, L_{n-2}, L_{n-1} = \varepsilon, L_n = \varnothing$, ordered as in Remark 3. By Remark 4, all the quotients of the form $K_2 \cup L_i$ or $K_j \cup L_2$ must be reached by letters if the bound is to hold, and this is impossible if the size of the alphabet is smaller than the number of such quotients. □

5 Product and Star

The complexity of the product of prefix-free languages is $m + n - 2$ [18]. For suffix-free languages, the complexity is $(m-1)2^{n-1} + 1$ [17]. Since bifix-free languages are prefix-free, and the witness prefix-free languages a^{m-2} and a^{n-2} are also subword-free, we have the following result:

Theorem 4 (Product). *If K and L are bifix-free with $\kappa(K) = m$ and $\kappa(L)$, where $m, n \geq 2$, then $\kappa(KL) \leq m + n - 2$. Furthermore, there are unary subword-free languages that meet this bound.*

The complexity of star is n for prefix-free languages [18], and $2^{n-2} + 1$ for suffix-free languages [17]. We now extend these results to bifix-, factor-, and subword-free languages. The quotient of L^* by ε is $L^* = \varepsilon \cup LL^*$, and the following formula holds for a quotient of L^* by a non-empty word w [5]:

$$(L^*)_w = \Big(L_w \cup \bigcup_{\substack{w=uv \\ u,v \in \Sigma^+}} (L^*)_u^\varepsilon L_v\Big) L^*.$$

Theorem 5 (Star). *If L is bifix-free with $\kappa(L)$, where $n \geq 3$, then $\kappa(L^*) \leq n - 1$. Furthermore, there are binary subword-free languages that meet this bound.*

Proof. Assume that L is bifix-free. Then it is prefix-free, has only one final quotient, namely ε, and has the empty quotient, by Remark 1. Moreover, since L is suffix-free, the quotient L is uniquely reachable by ε, by Remark 2.

Let L_w be a non-empty quotient of L by a non-empty word w. Let us show that $(L^*)_u^\varepsilon = \varnothing$ for every proper non-empty prefix u of w. Assume for contradiction that $\varepsilon \in (L^*)_u$, where $w = uv$ for some non-empty words u and v. Then $u \in L^*$, and so there exist words x in L and y in L^* such that $u = xy$. This gives $L_w = L_{xyv} = \varepsilon_{yv} = \varnothing$ because $x \in L$ implies $L_x = \varepsilon$. This is a contradiction, and so we must have $(L^*)_u^\varepsilon = \varnothing$. Hence, if L_w is non-empty, then $(L^*)_w = L_w L^*$, by the equation above. Now if L_w is final, then $L_w = \varepsilon$, and so $(L^*)_w = L^* = (L^*)_\varepsilon$. There are $n-2$ choices for non-final and non-empty quotients L_w. But, for a non-empty word w, we have $L_w \neq L$ since L is uniquely reachable by ε. This reduces the number of choices to $n-3$ since $n \geq 3$.

Now consider $L_w = \varnothing$ for a non-empty word w. Let u be the longest proper non-empty prefix of w such that $(L^*)_u^\varepsilon = \varepsilon$. If no such u exists, then $(L^*)_w = \varnothing$. Otherwise, let us show that for every proper non-empty prefix u' of u, we must have $(L^*)_{u'}^\varepsilon = \varnothing$. Assume for a contradiction that $(L^*)_{u'}^\varepsilon \neq \varnothing$. Then $u' \in L^*$ and also $u \in L^*$. So there exist $x, x' \in L$ and $y, y' \in L^*$ such that $u = xy$ and $u' = x'y'$. Since u' is a proper prefix of u, one of x and x' is a prefix of the other. If $x \neq x'$, then L is not prefix-free, which is a contradiction. If $x = x'$, then $y \neq y'$ and y' is a proper prefix of y. By an induction on the length of y' we can derive a contradiction that L is not prefix-free. So $(L^*)_w = (L^*)_u^\varepsilon L_v L^* = L_v L^*$, which has already been counted.

In total, there are at most $n-1$ quotients of L^*. The subword-free language a^{n-2} over $\{a, b\}$ meets the bound since the language $(a^{n-2})^*$ has $n-2$ quotients of the form $a^{n-2-i}(a^{n-2})^*$ for $i = 1, 2, \ldots, n-2$, and it has the empty quotient. □

6 Reversal

The last operation we consider is reversal. In [17, 18] it was shown that the complexity of reversal is $2^{n-2}+1$ for suffix-free or prefix-free languages. We show that this bound can be reduced for bifix-free languages. We use the standard method of reversing the quotient dfa $\mathcal{D}$ of L to obtain an nfa $\mathcal{D}^R$ for the language L^R, and then we apply the subset construction to nfa $\mathcal{D}^R$ to get a dfa for L^R.

Theorem 6 (Reversal: Bifix- and Factor-Free Languages). *If L is a bifix-free language with $\kappa(L)$, where $n \geq 3$, then $\kappa(L^R) \leq 2^{n-3}+2$. Moreover, there exist ternary factor-free languages that meet this bound.*

Proof. If L is bifix-free, then so is L^R. Since L is prefix-free, it has exactly one final quotient, ε, and also has the empty quotient.

Consider the quotient automaton $\mathcal{D}$ for L, and remove the empty quotient and all the transitions to the empty quotient. Reverse this incomplete dfa to get an $(n-1)$-state nfa $\mathcal{D}^R$ for L^R. Consider the subset automaton of the nfa $\mathcal{D}^R$. The initial state of the subset automaton is the singleton set $\{f\}$, where f is the quotient ε in the quotient automaton $\mathcal{D}$. No other subset containing state f is reachable in the subset automaton since no transition goes to state f in nfa $\mathcal{D}^R$. This gives at most $2^{n-2}+1$ reachable states. However, language L^R is prefix-free, and so all the final states of the subset automaton accept only the empty word, and can be merged into one state. Hence $\kappa(L^R) \leq 2^{n-3}+2$.

If $n = 3$ or $n = 4$, then factor-free languages a and aa, respectively, meet the bounds.

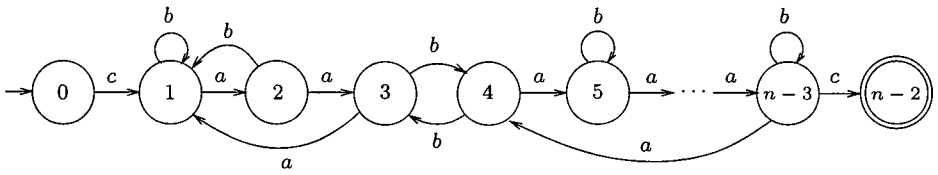


Figure 9: The ternary factor-free language meeting the bound $2^{n-3}+2$ for reversal.

If $n \geq 5$, then consider the language $L = cKc$, where K is a regular language over the alphabet $\{a, b\}$ with $\kappa(K) - 3$ meeting the upper bound 2^{n-3} for reversal [26, 39]. The quotient automaton of L without the empty state is shown in Figure 9.

By Proposition 1, language L is factor-free, and $\kappa(L)$. Since $\kappa(K^R) = 2^{n-3}$, there exists a set S of 2^{n-3} words over $\{a, b\}$ that define distinct quotients of language K^R. Then the quotients of cK^Rc by $2^{n-3}+2$ words ε, cw with $w \in S$, and cuc for some word u in K^R, are distinct as well. This gives $\kappa(L^R) = 2^{n-3}+2$. □

Theorem 7 (Reversal: Subword-Free Languages). *If L is a subword-free language over an alphabet Σ with $\kappa(L)$, where $n \geq 3$, then $\kappa(L^R) \leq 2^{n-3}+2$. The bound is tight if $|\Sigma| \geq 2^{n-3}-1$, but cannot be met for smaller alphabets. The bound cannot be met if L contains a word of length at least 3.*

Proof. Suppose L is a subword-free language. Let $\mathcal{D} = (Q, \Sigma, \delta, s, \{f\})$ be the quotient dfa of L with $Q = \{s, q_1, \ldots, q_{n-3}, f, e\}$ as the state set, where e and f correspond to the quotients $\varnothing$ and ε. Construct an nfa $\mathcal{D}^R$ for L^R, and consider the corresponding subset automaton.

The initial state of the subset automaton is $\{f\}$, and no other state contains f. Next, all the states containing s can be merged. As in Theorem 6, we get at most $2^{n-3}+2$ reachable states. If $\kappa(L^R) = 2^{n-3}+2$, then the set $\{q_1, q_2, \ldots, q_{n-3}\}$ must be reachable. Therefore there must exist a non-empty word v such that, for all q_i, we have $\delta(q_i, v) = f$. Now suppose there exists a word w in L such that $|w| > 2$. Let $w = abx$ where $a, b \in \Sigma$ and $x \in \Sigma^+$. Also suppose $\delta(s, a) = q_i$ and $\delta(q_i, b) = q_j$. Then we have $av, abv \in L$, showing that L is not subword-free, which is a contradiction. Hence, if any word in L has length at least 3, then $\kappa(L^R) < 2^{n-3}+2$. Now note that, if all the words in L have length at most 2, the only possible quotients of L^R are L^R, $(L^R)_a$ for all $a \in \Sigma$, ε, and $\varnothing$. Therefore $\kappa(L^R) \leq |\Sigma| + 3$, and the second claim follows.

Now consider tightness. If $n = 3$, then the bound is met by the unary subword-free language a. Let $n \geq 4$ and $\ell = 2^{n-3} - 1$. Also let $\Sigma = \{a_1, a_2, \ldots, a_\ell\}$, and let $S_1, S_2, \ldots, S_\ell$ be all the non-empty subsets of $\{1, 2, \ldots, n-3\}$. Now let

$$L^R = a_1\Big(\bigcup_{j \in S_1} a_j\Big) \cup a_2\Big(\bigcup_{j \in S_2} a_j\Big) \cup \cdots \cup a_\ell\Big(\bigcup_{j \in S_\ell} a_j\Big).$$

Since L^R only contains two-letter words, languages L^R and L are subword-free. The quotients of L^R are L^R, ε, $\varnothing$, and $(L^R)_{a_i} = \bigcup_{j \in S_i} a_j$ for $i = 1, 2, \ldots, \ell$.

Therefore $\kappa(L^R) = l + 3 = 2^{n-3} + 2$. But for L, the only possible and distinct quotients are L, ε, $\varnothing$, and L_{a_i} for $i = 1, 2, \ldots, n-3$. Thus $\kappa(L)$. □

7 Conclusions

Our results are summarized in Tables 1 and 2, where "B-, F-free" stands for bifix-free and factor-free, and "S-free" for subword-free. The bounds for operations on prefix-free languages are from [17, 21], on suffix-free languages from [13, 18, 25], and on regular languages from [31, 33, 32, 42]. For languages over a unary alphabet $\Sigma = \{a\}$, the concepts prefix-, suffix-, factor-, and subword-free coincide, and L is xfix-free with $\kappa(L)$ if and only if $L = \{a^{n-2}\}$.

In the case of subword-free languages the size of the alphabet cannot be decreased. In the other cases, whenever the size of the alphabet is greater than 2, we do not know whether or not the bounds are tight for smaller alphabets.

	$K \cup L, K \oplus L$	$\lvert\Sigma\rvert$	$K \cap L$	$\lvert\Sigma\rvert$	$K \setminus L$	$\lvert\Sigma\rvert$
free unary	$\max(m, n)$		m if m, 1 otherwise		m if $m \neq n$, 1 otherwise	
prefix	$mn - 2$	2	$mn - 2(m + n - 3)$	2	$mn - (m + 2n - 4)$	2
suffix	$mn - (m + n - 2)$	2	$mn - 2(m + n - 3)$	2	$mn - (m + 2n - 4)$	2
B-, F-free	$mn - (m + n)$	3	$mn - 3(m + n - 4)$	2	$mn - (2m + 3n - 9)$	2
S-free	$mn - (m + n)$	s_1	$mn - 3(m + n - 4)$	s_2	$mn - (2m + 3n - 9)$	s_3
regular	mn	2	mn	2	mn	2

Table 1: Complexities of Boolean operations on xfix-free languages; $s_1 = m + n - 3$, $s_2 = m + n - 7$, $s_3 = m + n - 6$.

	KL	$\lvert\Sigma\rvert$	L^*	$\lvert\Sigma\rvert$	L^R	$\lvert\Sigma\rvert$
free unary	$m + n - 2$		$n - 2$		n	
prefix-free	$m + n - 2$	1	n	2	$2^{n-2} + 1$	3
suffix-free	$(m - 1)2^{n-1} + 1$	3	$2^{n-2} + 1$	2	$2^{n-2} + 1$	3
B-, F-free	$m + n - 2$	1	$n - 1$	2	$2^{n-3} + 2$	3
S-free	$m + n - 2$	1	$n - 1$	2	$2^{n-3} + 2$	$2^{n-3} - 1$
regular	$(2m - 1)2^{n-1}$	2	$2^{n-1} + 2^{n-2}$	2	2^n	2

Table 2: Complexities of product, star, and reversal on xfix-free languages.

References

[1] Ang, T. and Brzozowski, J. Languages convex with respect to binary relations, and their closure properties. *Acta Cybernet.*, 19(2):445–464, 2009.

[2] Bassino, F., Giambruno, L., and Nicaud, C. Complexity of operations on cofinite languages. In López-Ortiz, A., editor, *Proceedings of the 9th Latin American Theoretical Informatics Symposium, (LATIN)*, volume 6034 of *LNCS*, pages 222–233. Springer, 2010.

[3] Berstel, J., Perrin, D., and Reutenauer, C. *Codes and Automata (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, 2010.

[4] Brzozowski, J. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[5] Brzozowski, J. Quotient complexity of regular languages. *J. Autom. Lang. Comb.*, 15(1/2):71–89, 2010.

[6] Brzozowski, J. In search of the most complex regular languages. *Int. J. Found. Comput. Sc.*, 24(6):691–708, 2013.

[7] Brzozowski, J., Jirásková, G., and Li, B. Quotient complexity of ideal languages. *Theoret. Comput. Sci.*, 470:36–52, 2013.

[8] Brzozowski, J., Jirásková, G., Li, B., and Smith, J. Quotient complexity of bifix-, factor-, and subword-free regular languages. In Dömösi, P. and Szabolcs, I., editors, *Automata and Formal Languages, (AFL 2011)*, pages 123–137. Institute of Mathematics and Informatics, College of Nyíregyháza, Hungary, 2011.

[9] Brzozowski, J., Jirásková, G., and Zou, C. Quotient complexity of closed languages. *Theory Comput. Syst.*, 54:277–292, 2014.

[10] Brzozowski, J. and Liu, B. Quotient complexity of star-free languages. *Internat. J. Found. Comput. Sci.*, 26(6):1261–1276, 2012.

[11] Brzozowski, J. and Szykuła, M. Large aperiodic semigroups. `http://arxiv.org/abs/1401.0157`, Dec 2013.

[12] Câmpeanu, C., Culik II, K., Salomaa, K., and Yu, S. State complexity of basic operations on finite languages. In Boldt, O. and Jürgensen, H., editors, *Revised Papers from the 4th International Workshop on Automata Implementation, (WIA)*, volume 2214 of *LNCS*, pages 60–70. Springer, 2001.

[13] Cmorik, Roland and Jirásková, Galina. Basic operations on binary suffix-free languages. In Kotásek et al. [29], pages 94–102.

[14] Eom, H-S., Han, Yo-S., and Jirásková, G. State complexity of basic operations on non-returning regular languages. In Jürgensen and Reis [28], pages 54–65.

[15] Haines, L. H. On free monoids partially ordered by embedding. *J. Combin. Theory*, 6(1):94–98, 1969.

[16] Han, Yo-S. and Salomaa, K. State complexity of union and intersection of finite languages. *Internat. J. Found. Comput. Sci.*, 19(3):581–595, 2008.

[17] Han, Yo-S. and Salomaa, K. State complexity of basic operations on suffix-free regular languages. *Theoret. Comput. Sci.*, 410(27-29):2537–2548, 2009.

[18] Han, Yo-S., Salomaa, K., and Wood, D. Operational state complexity of prefix-free regular languages. In Ésik, Z. and Fülöp, Z., editors, *Automata, Formal Languages, and Related Topics*, pages 99–115. University of Szeged, Hungary, 2009.

[19] Iván, S. On state complexities of unions of binary factor-free languages. `http://arxiv.org/abs/1405.1107`, 2014.

[20] Jirásek, J. and Jirásková, G. Cyclic shift on prefix-free languages. In Bulatov, A. A. and Shur, A. M., editors, *CSR*, volume 7913 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2013.

[21] Jirásková, G. and Krausová, M. Complexity in prefix-free regular languages. In McQuillan, I., Pighizzini, G., and Trost, B., editors, *Proceedings of the 12th International Workshop on Descriptional Complexity of Formal Systems (DCFS)*, pages 236–244. University of Saskatchewan, 2010.

[22] Jirásková, G. and Masopust, T. Complexity in union-free regular languages. *Int. J. Found. Comput. Sci.*, 22(7):1639–1653, 2011.

[23] Jirásková, G. and Masopust, T. On the state complexity of the reverse of $\mathcal{R}$- and $\mathcal{J}$-Trivial regular languages. In Jürgensen and Reis [28], pages 136–147.

[24] Jirásková, G. and Nagy, B. On union-free and deterministic union-free languages. In Baeten, J. C. M., Ball, T., and de Boer, F. S., editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 179–192. Springer, 2012.

[25] Jirásková, G. and Olejár, P. State complexity of union and intersection of binary suffix-free languages. In Bordihn, H., Freund, R., Holzer, M., Kutrib, M., and Otto, F., editors, *Proc. of the Workshop on Non-Classical Models for Automata and Applications (NCMA)*, pages 151–166. Austrian Computer Society, 2009.

[26] Jirásková, G. and Šebej, J. Reversal of binary regular languages. *Theor. Comput. Sci.*, 449:85–92, 2012.

[27] Jürgensen, H. and Konstantinidis, S. Codes. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*, pages 511–607. Springer, 1997.

[28] Jürgensen, H. and Reis, R., editors. *Descriptional Complexity of Formal Systems - 15th International Workshop, DCFS 2013, London, ON, Canada, July 22-25, 2013. Proceedings*, volume 8031 of *Lecture Notes in Computer Science*. Springer, 2013.

[29] Kotásek, Zdenek, Bouda, Jan, Cerná, Ivana, Sekanina, Lukás, Vojnar, Tomás, and Antos, David, editors. *Mathematical and Engineering Methods in Computer Science - 7th International Doctoral Workshop, MEMICS 2011, Lednice, Czech Republic, October 14-16, 2011, Revised Selected Papers*, volume 7119 of *Lecture Notes in Computer Science*. Springer, 2012.

[30] Krausová, M. Prefix-free regular languages: Closure properties, difference, and left quotient. In Kotásek et al. [29], pages 114–122.

[31] Leiss, E. Succinct representation of regular languages by boolean automata. *Theoret. Comput. Sci.*, 13:323–330, 2009.

[32] Maslov, A. N. Estimates of the number of states of finite automata. *Dokl. Akad. Nauk SSSR*, 194:1266–1268 (Russian)., 1970. English translation: Soviet Math. Dokl. **11** (1970) 1373–1375.

[33] Mirkin, B. G. On dual automata. *Kibernetika (Kiev)*, 2:7–10 (Russian)., 1966. English translation: Cybernetics **2** (1966) 6–9.

[34] Perrin, D. Finite automata. In van Leewen, J., editor, *Handbook of Theoretical Computer Science*, volume B, pages 1–57. Elsevier, 1990.

[35] Pighizzini, G. and Shallit, J. Unary language operations, state complexity and Jacobsthal's function. *Internat. J. Found. Comput. Sci.*, 13:145–159, 2002.

[36] Shyr, H. J. *Free Monoids and Languages*. Hon Min Book Co, Taiwan, 2001.

[37] Shyr, H. J. and Thierrin, G. Hypercodes. *Inform. and Control*, 24:45–54, 1974.

[38] Thierrin, G. Convex languages. In Nivat, M., editor, *Automata, Languages and Programming*, pages 481–492. North-Holland, 1973.

[39] Šebej, J. Reversal of regular languages and state complexity. In Pardubská, D., editor, *Proc. 10th ITAT*, pages 47–54. Šafárik University, Košice, 2010.

[40] Yu, S. Regular languages. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1997.

[41] Yu, S. State complexity of regular languages. *J. Autom. Lang. Comb.*, 6:221–234, 2001.

[42] Yu, S., Zhuang, Q., and Salomaa, K. The state complexities of some basic operations on regular languages. *Theoret. Comput. Sci.*, 125:315–328, 1994.

Received 27th December 2013

Acta Cybernetica **21** (2014) 529–552.

Mobile Platforms and Multi-Mobile Platform Development*

Hassan Charaf, Péter Ekler, Tamás Mészáros, Imre Kelényi, Bence Kovari, István Albert, Bertalan Forstner, and László Lengyel†

Abstract

Mobile devices and mobile applications have a significant effect on the present and on the future of the software industry. The diversity of mobile platforms necessitates the development of the same mobile application for all major mobile platforms, which requires considerable development effort. Mobile application developers are multiplatform developers, but they prioritize the platforms, therefore, not all platforms are equally important for them. Appropriate methods, processes and tools are required to support the development in order to achieve better productivity. The main motivation of our research activity is to provide a method, which increases the development productivity and the quality of the applications and also reduces the time to market. The paper discusses our model-driven results on the field of multi-mobile platform development.

Keywords: Design Tools and Techniques, Domain-specific architectures, Domain engineering, Reusable libraries, Software Engineering Process

1 Introduction

Mobile devices play a significant role in the daily lives of the majority of people living in a consumer-based society [12] [32]. Many people own one or more mobile devices, from numerous device distributors, with a variety of special features, capabilities and application programming frameworks. The diversity of the mobile platforms requires to develop the same application several times, once for each of the supported mobile platforms.

*This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfred. This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

†Budapest University of Technology and Economics, E-mail: {hassan, peter.ekler, mesztam, imre.kelenyi, kovari, ialbert, bertalan.forstner, lengyel}@aut.bme.hu

In 2008 $4.2 billion was spent on mobile applications. In 2013, an estimated $29.5 billion will be spent [12]. Both the tendency and the magnitude of these numbers reflect the fact that mobile phones are a strong part of our everyday life. We always take our mobile phones with us, continuously check our e-mails, social networks and other websites.

The mobile developer mindshare in 2013 shows that Android is leading with more than 70% of developers using the platform, followed by iOS at about 55% [33]. The [33] survey is based on more than 6,000 developers' mind from over 115 countries. Currently HTML5 is also a mobile development technology, with more than 50% of the developer population using HTML5 technologies for developing mobile applications. In case of HTML5, there are different approaches to mobile development:

- Mobile websites: websites that are designed to be rendered on small screens. Responsive websites are included into this category.
- Mobile web applications: websites with offline storage and deeper browser integration.
- Hybrid applications, using native wrapper: in this case HTML code wrapped in a browser, within a native shell (e.g. PhoneGap [26]).
- Applications using native JavaScript API: platforms exposing software and hardware services through a JavaScript API (e.g. Firefox OS [9] and BlackBerry 10).
- HTML5/JavaScript applications converted to native: JavaScript is handled as a platform independent code and convert it to a native application (e.g. Appcelerator Platform).

The applications developed with the first two approaches are distributed as web applications, while others as native applications via the application stores.

The latest research [33] shows that developers' platform choices depend very much on the goal they aim to achieve. When it comes to platform selection, contract developers vote for platforms that will generate more revenue, Chief Information Officers (CIOs) prefer efficiency and low cost, Chief Marketing Officers (CMOs) focus on reach, while hobbyists want to experiment with newer platforms [34].

There is no one-size fits all across mobile platforms. iOS is selected more frequently than average by developers who focus on value revenue potential, graphics, application discovery and user reach. Developers tend to use HTML5 more frequently as their primary platform when they value porting and speed and cost of development. BlackBerry 10 is used more frequently than average as a primary platform by developers valuing developer community programs. Windows Phone is most popular for developers who are already familiar with the .NET Framework [33].

We can see that the current market is rather colorful. Today's mobile application developer is a multi-platform developer. Developers use almost 3 mobile platforms

concurrently. The surveys do not show a significant difference in the last 3 years, i.e. the value is always between 2.6 and 3.2. Therefore, the main motivation of our research activities is to support the developers with appropriate multi-platform development methods and tools.

Our team has a reasonable experience both on the filed of mobil application design and development, and on the multi-platform management. In our approach, multi-platform solutions are driven by model-based solutions and software artifact generation methods. During the last 12 years we have supported multi-mobile application development in different ways with several different methods. We have worked out common mobile platforms, provided methods to synchronize user interfaces of different mobile platforms [20], and applied multi-paradigm modeling techniques in multi-platform mobile development [17]. Furthermore, we have moved forward a model-based unification of mobile platforms method [18]. Also, we have introduced the VMTS mobile toolkit [19] that is based on out modeling and model processing framework (Visual Modeling and Model Transformation System [4] [35]).

In the last two years we have standardized the results of all these activities, redesigned and rebuilt our multi-platform development method. This paper summarizes the result of our actual activities and introduces the new method we use in our current multi-platform development software projects.

The rest of this paper is organized as follows. Section 2 discusses the mobile platforms, concentrating on the market share and platform diversities. Section 3 provides our earlier results on the field of multi-mobile platform development. Section 4 and 5 introduces the details of the actual results and current multi-mobile platform development method. The discussed model-driven method supports the generation of software artifacts from common mobile application models. Section 6 summarizes the related work and compares our solution with other multi-platform approaches. Finally, conclusions are elaborated.

2 Mobile platforms

The different mobile platforms and the different strategies of the device manufacturers continuously modify the market conditions. The manufacturers provide not only devices but different services as well. End users are served with several custom solutions and trendy features. The competition of the platforms and the handset makers led to the current situation.

The mobile device market currently (in 2013) is dominated by Android devices, from low-end feature-phone replacements to high-end devices. In the first half of 2013, Android had about 75% of all smartphone shipments, while iOS had another 18% leaving very little room for anyone else [13] [33].

Smartphone sales by handset makers show that Samsung is at the top end, and there are numerous competitors at the long tail of the distribution (Apple, LG, Nokia, Huawei, BlackBerry, ZTE). The other segment, which is not covered by the mentioned manufacturers of smartphone makers is currently selling as many devices as Samsung. These handset makers are made up of hundreds of Android

handset producers. Taking these into account, it is interesting that Samsung's main competitor in terms of market share is not Apple, but these handset makers who are able to supply the cheapest possible smartphones, customized for every corner of the developing world [33].

The reason Samsung is still making profits among modular handset makers is because they have realized that there are no profits to be made in handset production itself. In other words, hardware is not enough. Instead, value has migrated upwards in the technology stack (to services) and downwards (to handset components). Also, it is a fact that Samsung spends more for marketing than Coca-Cola.

The lead platform is where new applications and features are first rolled out and which can be the star of the marketing launch. Prioritization also has an impact on focus, application quality, sales and revenue. The way developers prioritize the platforms has a direct impact on the overall perception of the platform. If most developers treat a platform as a second-class citizen, this will reflect negatively on the application quality and consequently, on developers' revenue opportunity on that platform. Developers that prioritize a platform will act as evangelists for that platform, as they are likely to create high quality, most up-to-date applications and praise the platform at events or social media. As a consequence, supporting developers with tools, documentation, frameworks and easy-to-use libraries can increase the market share of the platform [33].

Actual surveys show that more than 80% of mobile developers are using iOS, Android or HTML5 (mobile) as their primary target platform. In this world, not all platforms are equally important to a developer. Platform priorities also depend on the level of experience. Developers who are fresh to mobile have a much stronger preference towards Android, with almost twice as many new mobile developers preferring Android than iOS.

Asking a developer to switch to a different platform is like asking someone to learn a foreign language. This is a task that takes months. The challenge is not just about the language (Objective C, Java, C#, HTML or JavaScript) itself. It is the set of APIs, development environment, publishing process, and the 3rd party tools ecosystem that supports the platform. Learning a language nowadays is not very hard, they all look the same (literally), but what a developer has to learn is the API.

Developer tools are not just nice-to-have. Tools are in the must-have application development arsenal of the most sophisticated developers, and also those making most revenues. Appropriate tools can increase the productivity also the quality of the products. Tools can support to utilize certain artifacts for several different mobile platforms. Developer tools can make a platform more attractive for developers. Also the tool can reduce the time to market for mobile applications.

As a summary we can conclude that today's developers are multiplatform developers, but not all platforms are equally important for them. In case of the multiplatform developers the main decisions are often based on priorities. Appropriate methods, processes and tools are required to target several platforms, support the development in order to achieve better productivity and quality.

3 Multi-mobile platform development - Our earlier results

In embedded software development, reuse is recognized as a key factor for better productivity at lower costs. In the ideal case, the product family approach makes it possible to reuse elements in a whole range of related products. The family members are based on a common architecture and rely heavily on reusable components, thus allowing the developers to concentrate only on the required variation between the products. This approach also enables the developers to focus on design instead of implementation details.

We have realized that from the perspective of mobile software development one of the greatest challenges is caused by the fragmentation of mobile platforms. Each mobile platform has different advantages, thus it is hard to find an ultimate platform for applications. Not only the development tools but the supported languages and even the application life cycles are different, thus each platform requires developers with special knowledge related to the specific platform. To make it possible to support the creation of a common development platform, we have to apply methods that move a reasonable part of the development to a higher abstraction level.

One of our first solution was the **Common Mobile Platform** (CMP), which was a solution applied between 2005 and 2008 to model mobile applications and generate source code for different mobile platforms. CMP defines a model and XML language for describing mobile applications and provide a generator mechanism which generates working source code for the following mobile platforms: Symbian 3rd edition, 5th edition [14], Windows Mobile 5-6, Microsoft .NET Compact Framework (.NET CF) [37] and Java 2 Micro Edition (J2ME) [16]. The solution was based on the Model-Driven Architecture (MDA) [22]. We differentiated platform-independent models that described the common behavior of the required application and platform-specific models generated automatically from platform-independent models. The source code is also automatically generated from platform-specific models. The generated code utilized the frameworks prepared to support the area.

Domain-specific models [11] have another remarkable advantage over usual software development methods: by using different code generation templates, we can produce applications for different application platforms. This means that from a single model set, we can generate our application for all target platforms, including mobile platforms, web and desktop applications as well, and thus, changes applied on the models can be immediately implemented on all platforms.

The Simplian Framework. Even tough, Symbian was one of the most popular mobile platforms, Symbian-based software development was far more difficult and required more specific skills than the development of desktop applications. This stemmed not only from the prestandard C++ characteristics, but also from the absence of easy-to-use integrated development tools.

We can mention, for example, the complicated memory management (for instance cleanup stack and two-phase object construction), the special exception-handling (leave-mechanism), string- and array handling, or the unique aspects of

resource-management. Also, developers must strictly take care of these uncommon circumstances, since ignoring them could lead to bugs that are hard to identify.

We provided a class library that helped the programmer by hiding all the recurring tasks deriving from the mentioned facts. The Simplian Framework provided an API for constructing the user interface of the application, and other simple means to bind data to the widgets, or send and receive them through different communication channels. Simplian also provided tools to generate the C++ code from a well-defined, platform-independent XML file, which could be constructed by the modeling tool [1].

The Symbian platform related model processor supported user interface, data binding and database generation, thus, a database metamodel was also provided as an input. The model processing solution used the Microsoft CodeDOM technology [31] for code generation. The CodeDOM consisted of classes representing the syntactic elements of the .NET languages, like C# and managed C++.

The .NET Compact Framework. Applying the same method with different model processors we generated applications from the same models for devices with .NET Compact Framework. We do not introduce the model processors related to all aspects of different mobile platforms, but we note that the main difference between the two transformations (Symbian and .NET CF related transformations) was in the resource model (user interface model) processing. For the .NET CF platform certain properties of the user controls are generated based on the attributes of the resource model. The rewriting rules did not use these attribute values during the generation for Symbian platform, because the controls were placed strictly one below other.

The presented approach made possible to use visual languages to define user interface, database and communication models that described the different aspects of mobile applications. The solution provided model processors to generate the platform-specific source code. The solution was realized with domain-specific language engineering and graph rewriting-based model transformation.

4 Multi-mobile platform development - Overview of the actual method

Model-driven software engineering is an actively researched field. The growing size and complexity of software systems made software modeling technologies essential in application development. Model-driven development approaches can increase the development productivity and the quality of the produced software artifacts.

Model-driven development approaches emphasize the use of models at all stages of system development. In model-based development, models are used to describe the most artifacts of the system, i.e., interfaces, interactions, and properties of all the components that comprise the system. These models can be manipulated in a number of different ways to describe the system, and in certain cases to generate the complete implementation of the system. In order to capture the semantics that is as close as possible to the domain of the developed system in an effective manner,

building a domain-specific modeling language is a suitable choice. Using domain concepts to modeling systems helps increase productivity, makes systems easier to maintain and evolves and shortens the development cycle.

Our modeling and model transformation framework is the Visual Modeling and Transformation System (VMTS) [4] [35]. VMTS is a metamodeling environment which supports editing models according to their metamodels. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary").

Also, VMTS is a model transformation system, which transforms models using both template-based and graph rewriting techniques. Moreover, the tool facilitates the verification of the constraints specified in the transformation step during the model transformation process. VMTS has been developed since 2003.

The VMTS approach uses a graphical notation for control flow (the execution sequence of the transformation rules): stereotyped UML activity diagram [23]. The control flow language can express a transformation as an ordered sequence of the transformation rules. Classical graph grammars apply any production that is feasible. This technique is appropriate for generating and matching languages, but model-to-model transformations usually need to follow an algorithm that requires a stricter control over the execution sequence of the steps, with the additional benefit of making the implementation more efficient. The control flow language supports the following constructs: sequencing transformation steps, branching with C# conditions, hierarchical steps, parallel execution of the steps, and iteration.

The architecture of the application generation process is depicted in Figure 1. The modeling of mobile applications and the processing of these models are performed in a framework. We use mobile, domain-specific languages to define the required structure and application logic. Platform-specific model processors are applied to generate the executable artifacts for different target mobile platforms. The generated code is based upon the previously assembled mobile, platform-specific frameworks. These frameworks provide energy efficient solutions for mobile applications. Furthermore, some data processing or computationally intensive tasks are passed into the cloud to save the battery power of the mobile device.

Mobile, domain-specific languages address the connection points and commonalities of the most popular mobile platforms. These commonalities are the basis of further modeling and code generation methods. The main areas, covered by these domain-specific languages, are the static structure, business logic (dynamic behavior), database structure and communication protocol. Using these textual and visual languages, we are able to integrate the use of cloud services into the business logic.

For each target platform, a separate transformation should be realized since, at this step, we convert the platform-independent models into platform-specific executable code. The transformation expects the existence of the aforementioned frameworks and utilizes their methods. The generated source code is essentially a list of parameterized activities (commands) that certain functions of the mobile application should perform. This means that the core realization of the functions is not generated but utilized from mobile-platform specific frameworks, i.e., the

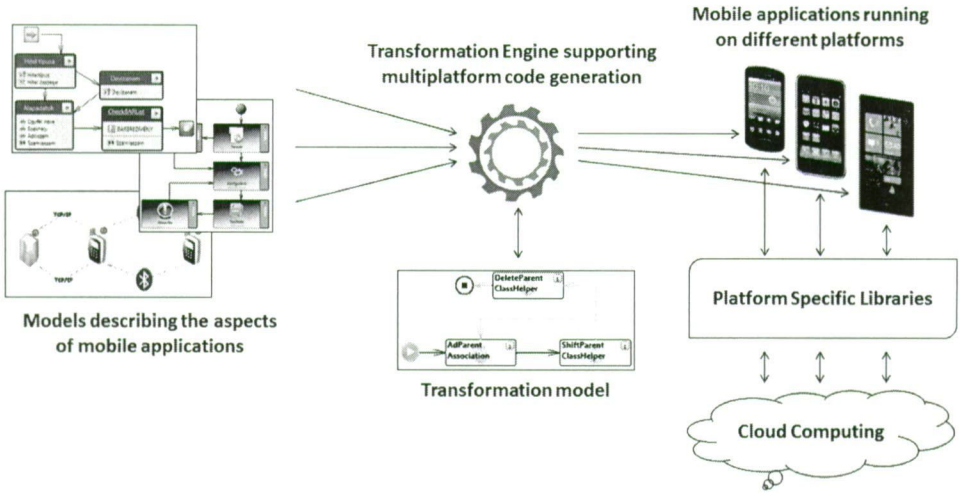


Figure 1: Supporting multi-mobile platform development

generated code contains the correct function calls in an adequate order, and with appropriate parameters. The advantages of this solution are the followings:

- The software designer has easier task. The model processors are simpler. The model processing is quicker. The generated source code is shorter and easier to read and understand.

- We use prepared mobile, platform-specific frameworks. These frameworks are developed by senior engineers of the actual mobile platform.

The current wave of our research activities is focused on the Android [3], iOS [15] and Windows Phone 8 [10] [29] platforms.

Model processors generate platform-specific source code. Developers integrate the generated code into the source of a native application. The generated code is based on well tested libraries created by platform specialists. Therefore, the generated code does not contain the whole implementation of the features, instead, it utilizes the services of the library via API calls. In this way the library can be utilized several times by different generated and hand written source code snippets. This is because the goal is not to generate all of the code, but (i) to perform the modifications where it is easier (either in the model world or in the source code), (ii) automatic synchronization from model to code, which is supported by separated source files for generated and hand written code (partial classes, inheritance), and to (iii) speed up the development of the same application for different platforms.

Table 1 summarizes the tasks that are typically required during mobile application development. The table highlights which tasks are supported with our approach and which tasks requires further manual coding. In the current state of

our solution the network communication and the resource management is already implemented in the framework while the others are under development.

Table 1: Mobile development tasks.

Task	Approach
User interface and screen flow	Manual (mockup can be generated)
Custom views	Manual
Persistence and data model	Generate
Network communication	Generate
Resources (localization)	Generate
Business logic	Generate
Multimedia (e.g.: camera, music)	Generate
Location based services and map	Generate

The key points and the evaluation of the method. In summary, we still do not believe that manual coding can be eliminated, because there are complex functions and platform-specific tasks, which require manual coding. This means that the gluing code required to integrate the generated source, platform specialties and the custom logic are manually added. But it is worth to model several parts of the application and generate the appropriate source code based on it. This increases the quality and shortens the time to market of the applications. We are applying the method in both mobile application development and server side component development. Key points of the approach are the prepared platform-specific frameworks and libraries. They can be utilized by both generated and handwritten code, therefore they support rapid development and provide higher quality.

5 The methods of multi-mobile platform development

5.1 Modeling: Defining the Mobile Applications on a Higher Abstraction Level

Instead of discussing all aspects of mobile platforms that the method supports, we decided to focus on one area of the covered fields and provide more details about it. The selected area is the rapid prototyping of REST-based communication channels (Representational State Transfer) [8] [27]. The methods supporting this area, i.e. the way how we model some aspects of the mobile platforms and how the model processors generate the source code, are similar for several different aspects of mobile platforms as well.

Depending on the expectations of the customer, the mobile application developers need to develop either only the mobile clients for an existing server application

or the server applications supporting the clients as well. In the most common scenario, there is one server-side application, and the mobile clients should be developed for multiple target platforms concurrently. Therefore, the communication layer should be developed for each platform. Though, the concrete implementations of the layers on the different platforms show significant similarity, it is not possible to reuse the source code between the platforms because of the different languages and runtime environments. These facts are motivating issues to apply platform-independent modeling and model processing to support these types of challenges.

Mobile applications usually apply a REST-based communication channel when it is about data exchange between the different devices. In REST, the operations of the server application can be accessed with the help of a properly formatted HTTP request. The parameters of the request may be encoded either into the request URL itself, or into the body of the HTTP request. In the latter case, the formatting of the parameters may be arbitrary, although the two most often applied serialization mechanisms for the parameter objects are XML and JSON. The server application responds to such a request with a HTTP response that contains the response parameters in its body (again, usually as XML or JSON). Even though we know if the request and response body is formatted as XML or JSON, their concrete schema may be arbitrary and is not tied to strict rules like in case of SOAP [30]. Consequently, the serialization and deserialization procedures as well as the URL generators and interpreters should be developed on both the client and the server sides. Furthermore, in case of the client application, the same development must be performed in case of each targeted mobile platform as well.

As an initial step, we focus on REST APIs exclusively and automate this error prone, monotonous coding for the client side. We have elaborated a modeling language that is able to describe server-side operations, the data types used and the way of parameter serialization. We have also prepared the generators that automate the creation of the client-side communication layer based on the models.

There are various forms to represent a modeling language. For practical reasons and to speed up the initial development we have chosen to realize this language based on an already existing programming language: C#. The idea is to define the communication API with the help of C# interfaces and to generate the concrete implementations based on these interface definitions for various platforms. Utilizing C# as a base language has several benefits over implementing a proprietary modeling language, or using general purpose data description languages like XML or JSON to describe the data models. First, the syntactic and semantic verification of the language can be performed using existing C# compilers without any additional effort. The appropriate usage of the types is forced by the strongly-typed property of C#. By compiling the interfaces into a .NET assembly, we can easily traverse and interpret the models by traversing the assembly using reflection. Next, since the final generated code is using languages similar to C# (C#, Java, Objective C), the data structures and interface definitions are close to the final implementation and can be described in a way familiar to the developers. Furthermore, the VMTS environment handles C# as the textual concrete syntax of the models, and

in a similar way visual concrete syntax can be also provided for the same model. Users can decide which concrete syntax (textual or visual) they want to use for the interface definition.

With the help of interfaces, we can precisely define the methods that can be called on the server application, and we can also define the possible data types we can pass to these methods or we can expect from these methods as a result value. The way how such a method call is performed, how the parameters are serialized when passing them to the methods and how the result values are deserialized on a successful call can be customized with the help of .NET attributes. The code generators processing these custom domain-specific models (interfaces) discover the attributes attached to the elements of an interface definition and modify the code generation accordingly.

Now we introduce our approach with an example, where the server has a simple method that is used to create a new user in the target system. The method expects one parameter (the name of the user) and returns nothing. The corresponding C# interface is the following.

```
[RestApi]
public interface MyService
{
    void insertUser(string userName);
}
```

To indicate that this interface defines the API of a server application we denote it with the *[RestApi]* attribute above the interface. By finding this attribute the code generator will recognize that this interface should be treated as a REST API interface, and it should generate the client-side proxy class for that. This *[RESTUrl]* attribute specifies which Url to invoke. This way, the generated proxy will navigate to the insertuser.php page, and pass the username as url parameter (like *insertuser.php?userName=XXXX*).

If we would like to use different parameter names instead of the names of the parameters in the C# interface, we may customize that using the *[RestParam]* attribute.

```
[RestApi]
public interface MyService
{
    [RestMethod(Url = "insertuser.php")]
    void InsertUser ([RestParam(Name="usr")]string userName);
}
```

In the example above, though, the name of the parameter is username, it is mapped to the *usr* http GET parameter (*insertuser.php?usr=XXXX*). If we need to use different HTTP methods, e.g. POST instead of GET to call the server-side service, we can specify it as the *CommandType* a parameter of the *[RestMethod]* attribute.

Changing the command type to POST (possible values are GET, POST, PUT, DELETE) we just instruct the code generator to generate a proxy code that uses the HTTP POST command to send the request. The passed parameters are still encoded into the request url, as before. If we would like to pass the parameter inside the body of the HTTP request as HTTP form parameter instead of the request url itself, then we can set it up using the *Mapping* parameter of the *[RestParam]* attribute.

```
[RestApi]
public interface MyService
{
    [RestMethod(Url = "insertuser.php",
       CommandType = CommandType.POST)]
    void InsertUser ([RestParam(Name="usr", Mapping =
       RestMethodMappingType.Body)]string userName);
}
```

The default value for Mapping is *RestMethodMappingType.Url.* Although, we can already pass single parameters both in the request URL and in the HTTP body as form parameters, often the argument should be handled as not a parameter of the target resource but a locator for the target resource. E.g. consider that the user we create is assigned to a specific client. But the client is not passed as a parameter to the insertuser.php page, but the insertuser.php page is located inside the appropriate client folder like *http://..../client1/insertuser.php...* . To map a specific parameter into the resource Url at a specific position, we must set the Mapping argument for that parameter to *RestMethodMappingType.Custom*, indicate the position of this parameter with the $ character.

Since a server method usually has a return value as well, it must be handled by the generated proxy code. Consider that the *InsertUser* method returns the unique id of the newly created user inside the HTTP response as plain text. This return value can simply be returned by the generated proxy method by setting the return type of the *InsertUser* method to string.

```
[RestApi]
public interface MyService
{
    [RestMethod(Url = "$client/insertuser.php", CommandType =
       CommandType.POST)]
    string InsertUser ([RestParam(Mapping =
       RestMethodMappingType.Custom)]string client,
       [RestParam(Name="usr",
       Mapping = RestMethodMappingType.Body)]string userName);
}
```

If the service returns a more complex value serialized in XML format, we may use custom return types as well, and indicate the type of serialization using the *ReturnFormat* property of the *RestMethod* attribute.

```
[RestApi]
public interface MyService
{
    [RestMethod(Url = "$client/insertuser.php", CommandType =
        CommandType.POST, ReturnFormat = FormatType.XML)]
    UserInfo InsertUser ([RestParam(Mapping =
        RestMethodMappingType.Custom)]string client,
         [RestParam(Name="usr", Mapping =
         RestMethodMappingType.Body)]string userName);
}
```

Here we support four options: *XML*, *JSON*, *Forms* and *Raw*. The Forms option covers the case when the parameters are formatted as HTTP POST key-value pairs, while the *Raw* formatting means transferring the value as it is, without any formatting.

The type of serialization and the way how parameters are passed is usually the same for each method within the same service. Thus, to avoid the tedious setup of the *RestParam* attribute at each method (if they differ from the default one), it is also possible to set up the common parameters for the entire service by placing this attribute above the interface declaration:

```
[RestApi]
[RestParam(Mapping = RestMethodMappingType.Body, Format =
FormatType.Form)]
public interface MyService
{
    [RestMethod(CommandType = CommandType.POST)]
    UserInfo InsertUser(string client, string userName);
}
```

In the example above, both parameters of the *InsertUser* method are serialized as form parameters inside the body of the HTTP request. If someone needs different behavior for specific method parameters, the default settings may be overridden by a *RestAttribute* parameter placed in from the related parameters.

Declaration of the custom data types. In most practical cases, the parameters expected by the methods or the return values of them are not only primitive types like string, integer or floating point number, but complex types consisting of multiple fields.

For this purpose, we have defined another interface-level attribute called *RestDto* that is the abbreviation of REST Data Transfer Object. Interfaces marked by this attribute will be handled by the code generator as simple classes used for representing and transmitting data. Of course, in addition to the standard data storing feature of such an object, the code generator may extend it with various additional features like change notification, equality comparison and so on.

Assume that when creating a new user, we would like to pass also the full name and the age of the user to be created, and we do not want to use a separate

method argument for them but handle them as one unit. Then we may wrap these parameters into a new DTO interface.

```
[RestDto]
public interface User
{
    string UserName { get; set; }
    string FullName { get; set; }
    int Age { get; set; }
}
```

A complex type cannot be simply encoded into the request Url, thus, we must set it up to be serialized inside the body of the HTTP POST request.

Using the default settings, the user parameter would be serialized as converting its fields into HTTP form parameters. But typically, in the REST communication rather XML or JSON serialization is applied. The way how complex parameters should be serialized can be specified with the *Format* argument of the *RESTParam* attribute.

```
[RestApi]
public interface MyService
{
    [RestMethod(Url = "$client/insertuser.php",
    CommandType = CommandType.POST)]
    string InsertUser ([RestParam(Mapping =
      RestMethodMappingType.Custom)]string client,
      [RestParam(Mapping = RestMethodMappingType.Body,
      Format = RESTFormatType.XML)]User user);
}
```

Of course, the same data may by serialized as XML in an infinite number of ways. By default, we assume each member field to be serialized as an XML tag identified by the name of the tag, while the value of the field is serialized as the content of the XML tag. If this value is of primitive type, then simply its printed value, if the value is of a complex type, then the same method is applied recursively. If we would like to change the way how the XML document is generated and parsed, we can perform it using the standard .NET XML formatter attributes by attaching them to the DTO definition.

Often, it is more comfortable to pass parameters to a method call as separate arguments, however, we would like to serialize them as one connected XML document. For example, we would like to use separate username, password and age arguments for the *InsertUser* method, but would like to serialize them in the request body as they would be part of one User class. For this purpose, you may set the IsCoupled property of the RestParam attribute to true:

```
[RestMethod(Url = "$client/insertuser.php",
```

```
CommandType = CommandType.POST)]
UserInfo InsertUser([RestParam(Mapping =
      RestMethodMappingType.Custom)]string client,
      [RestParam(Mapping = UrlMappingType.Body, Format =
      FormatType.XML, IsCoupled = true)]User user);
```

The method signature generated based on this declaration will look like this:

```
UserInfo InsertUser(string client, string userUserName,
      string userPassword, int userAge)
```

However, the last three parameters will be serialized as one XML document like

```
<User usr="joe" full="John Doe" age="30"/>
```

Figure 2 summarizes both the Service interface and the DTO specification attributes.

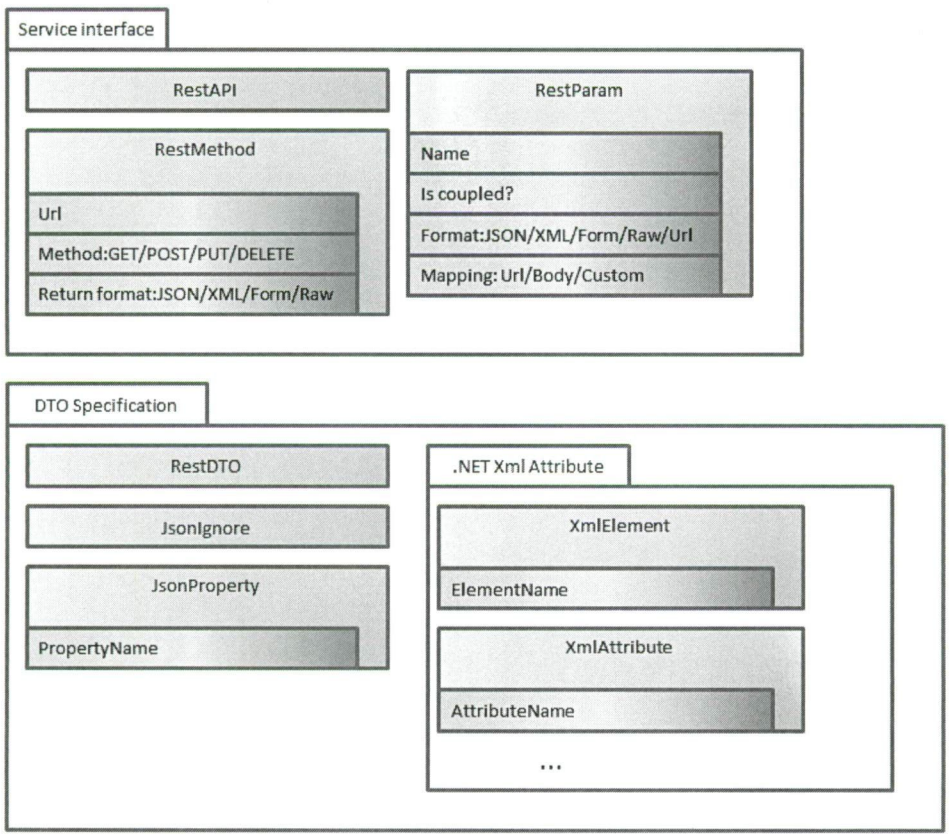


Figure 2: Service interface and DTO specification attributes

5.2 Model Processing: Generating the Software Artifacts

Having the features of a network service, including its methods and the applied data types already defined, the next step is that to generate executable source code which is able to perform the communication with the server component. There are various solutions that can be utilized when it is about code generation, we have chosen the Microsoft T4 (Text Template Transformation Toolkit) [24]. T4 is a mixture of static texts and procedural code: the static text is simply printed into the output while the procedural code is executed and it may result in additional texts to be printed into the output. Recall that, the interface definitions are compiled into a .NET assembly that can be loaded and traversed using reflection afterwards. The T4 templates we write also work on the reflected content of the assemblies.

Currently we are targeting two mobile platforms: Windows Phone 8 (C#) and Android (Java). Therefore, we need to prepare two different T4 templates for the two platforms. In case of Windows Phone, we expect the data transfer objects be represented by C# classes, the fields of the DTO entities should be represented as .NET properties, and the generated classes should also support some kind of change notification about changes in the properties. A possible implementation of the template is the following.

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute
  ("System.Runtime.Serialization", "3.0.0.0")]
[System.Runtime.Serialization.DataContractAttribute
  (Name="<#=type.Name#>")]
public partial class <#=type.Name#> :
  System.ComponentModel.INotifyPropertyChanged
{
<# foreach (var pi in type.GetProperties()) { #>

    private <#= pi.PropertyType.FullName #> <#=pi.Name#>Field;

    public <#= pi.PropertyType.FullName #> <#=pi.Name#>
    {
        get
        {
            return this.<#=pi.Name#>Field;
        }
        set
        {
<# if (!pi.PropertyType.IsValueType) { #>
          if (!object.ReferenceEquals(this.<#=pi.Name#>Field,
             value)) <# } else { #>
          if (!this.<#=pi.Name#>Field.Equals(value))<# } #>
          {
             this.<#=pi.Name#>Field = value;
```

```
            this.RaisePropertyChanged("<#=pi.Name#>");
          }
        }
    }
<#}#>

    public event System.ComponentModel.
      PropertyChangedEventHandler PropertyChanged;
    protected void RaisePropertyChanged(string propertyName)
    {
        System.ComponentModel.PropertyChangedEventHandler
          propertyChanged = this.PropertyChanged;
        if ((PropertyChanged != null))
        {
          PropertyChanged(this, new
            System.ComponentModel.PropertyChangedEventArgs(
            propertyName));
        }
    }
}
```

The input of the template (type) is the reflected DTO type. The resulting code will describe a partial class the name of which corresponds to the name of the interface. Then, the template iterates through all the fields declared in the interface, generates a private variable and a wrapper property for the variable. When setting up the value of the property, it checks if the new value is really different from the previous one, and changes the value of the underlying variable if it is really different. Then it also calls the *RaisePropertyChanged* method (passing the name of the changed property to it as parameter) that fires the *PropertyChanged* event.

For the Android (Java) implementation we do not need the DTOs to support any special features thus, here we just generate plain old Java (POJO) classes collecting private fields with getter and setter methods. The corresponding T4 template is much simpler as well.

```
<# Type type = this.DTOType; #>

<#= RenderCustomAttributes(type) #>
public class <#=type.Name#> {
<# foreach (var pi in type.GetProperties()) { #>

    <#= RenderCustomAttributes(pi) #>
    private <#= AHelper.MapType(pi.PropertyType) #> <#=pi.Name#>;

    public <#= AHelper.MapType(pi.PropertyType) #>
        get<#=pi.Name#>() { return this.<#=pi.Name#>;
```

```
    }

    public void set<#=pi.Name#>(<#=
        AHelper.MapType(pi.PropertyType) #> value) {
        this.<#=pi.Name#> = value;
    }
<#}#>
}
```

There is a big difference compared to generating target code for C#, though. Since the interface was also written in C# and uses the primitive types of the .NET Base Class Library, and we are traversing the .NET assembly, we must translate the .NET types to Java types. In case of the C# generator template, we could simply jump over this step, since we could use the same type names as in the interface definition itself (see *PropertyType.FullName* in the template). In case of the code template for Java, we perform this translation using the *TTHelper.MapType* method. This is a custom implementation handling some primitive types only, but can be arbitrarily extended with further types as well.

Proxy generation for the service methods. In general, it is advised to keep the generators as simple as possible, and outsource all the common implementations into helper classes or base classes. We followed this principle during realizing the code templates that generate the service proxy classes.

In case of the .NET implementation, all the communication-specific parts of the implementation are moved into the *RestClient* class. Its *SendRequest* method is used to perform the serialization and deserialization of the method parameters and the return value as well as the sending and receiving of the HTTP requests. For this purpose, the *SendRequest* method needs to know how the parameters should be processed during the assembling of the messages. Therefore, we utilize the same interface-, method- and parameter-level attributes into the generated code as well and let *SendRequest* discover these settings via reflection.

In case of the Android implementation, all the communication-specific parts of the implementation are outsourced into the *RESTTask* class that is subclassed from *AsyncTask*. In case of an Android *AsyncTask*, the network communication is performed on an asynchronous thread, and the caller of the thread is notified about the result via a *BroadcastReceiver* object. This object can then identify the called method and interpret the answer for the HTTP request. Therefore, we have split the implementation of the service proxy into two parts: the one is responsible for serializing the method parameters and instructing *RESTTask* to perform the HTTP request with the appropriate parameters, the other one (*RestBroadcastReceiver*) is responsible for receiving the asynchronous HTTP responses and to call the appropriate method to process the answer.

We use T4 templates to perform the above introduced method. Beside the discussed Android and Windows Phone platforms related templates the solution also supports the iOS platform with platform-specific templates.

5.3 Evaluation of the Solution

In summary, we can say that our objective targets one of the most pressing problems in the area of mobile software development, originating from the diversity of mobile platforms. To address this problem, we have provided a modeling language for mobile applications and developed frameworks for all platforms, which support the code generated from the models. The result is a method that allows the design of mobile applications and generates source code for major mobile platforms.

The presented domain-specific language that facilitates the definition of different aspects of the mobile application can have both textual and visual concrete syntax.

The main strength of the method is that it effectively supports the application design and development: speeds up the development and increases the code quality by automatically generating both the client and server side components. The generated code is not a full, buildable and executable artifact. It requires integration, i.e. further gluing code between the different mobile application aspects (user interface, communication, database management, others). This is a conscious decision behind the method. The objective is to support the repetitively occurring and lengthy coding processes and not to eliminate all aspects of programming. There are several areas that are easier to define with models and also several aspects that is more effective to directly write within the appropriate programming environment. The method helps to utilize both of these issues.

The method provides a general guidance that other teams can follow. The tool support still not available in a public form. We are continuously working on the tooling environment and will make it available for the community.

6 Related Work

This section introduces the most known cross-platform development frameworks and solutions. We summarize their capabilities and compare their achievements with our method.

PhoneGap [26] is a mobile development framework enabling developers to build applications for mobile devices using standards-based web technologies (HTML5, JavaScript and CSS3) instead of device-specific languages like Java, Objective-C or C#. The resulting applications are hybrid, meaning that they are neither truly native, because all layout rendering is done via web views instead of the platform's native UI framework, nor purely web-based, because they are not just web applications. Applications are packaged as applications for distribution and have access to native device APIs. It is possible to mix native and hybrid code snippets.

Earlier versions of PhoneGap required a developer making iOS applications to have an Apple computer, a developer making Windows Phone applications to have a computer running Windows, and so on. Currently, the PhoneGap Build service allows a programmer to upload his source code to a cloud compiler that generates applications for the supported platforms.

Appcelerator Titanium [2] is a platform that, similarly to PhoneGap, supports the development of mobile, tablet and desktop applications using web technologies. The Appcelerator Titanium framework is available since 2008.

Appcelerator Titanium Mobile framework allows web developers to apply existing skills to create native applications for iPhone and Android. However, in case of Appcelerator Titanium Mobile, developers should not only be familiar with web technologies and JavaScript syntax, but they also have to learn the Titanium API, which is different from familiar web frameworks like jQuery.

All application source code gets deployed to the mobile device where it is interpreted. Being interpreted means that some errors in the source code will not be caught before the program runs. Program loading takes longer than it does for programs developed with the native SDKs, as the interpreter and all required libraries must be loaded before interpreting the source code on the device can begin.

At the end of 2012, there were more than 30,000 applications shipped to the application stores built with Titanium. Appcelerator also offers cloud-based services for packaging, testing and distributing software applications developed on the Titanium platform.

Xamarin [36] is a company created by the engineers that created Mono [21] MonoTouch and Mono for Android, which are cross-platform implementations of the Common Language Infrastructure (CLI) and Common Language Specifications (Microsoft .NET).

Xamarin.Mobile is a library that exposes a single set of APIs for accessing common mobile device functionality across iOS, Android, and Windows platforms. The solution allows to use C# programing language and with the same code support all the mentioned platforms. Xamarin.Mobile currently abstracts the contacts, camera, and geo-location APIs across iOS, Android and Windows platforms. In the future, it will include notifications and accelerometer services.

Firefox OS [9] is a Linux-based open-source operating system for smartphones and tablet computers. It is being developed by Mozilla, the non-profit organization best known for the Firefox web browser. Firefox OS is designed to provide a complete community-based alternative system for mobile devices, using open standards and approaches like HTML5 applications, JavaScript, a robust privilege model, open web APIs to communicate directly with cellphone hardware and application marketplace.

As such, it competes with proprietary systems like Apple's iOS, Google's Android, and Microsoft's Windows Phone as well as other upcoming open source systems under development. Firefox OS was publicly demonstrated in February 2012.

Comparing our approach with PhoneGap, Appcelerator, Xamarin.Mobile and other multi mobile platform solutions, we can say that the goal is similar but not exactly the same. Available solutions target to produce the final executable files, i.e. the applications that are ready to use, that can be downloaded and installed. This approach is quite comfortable from both the end users and the developers point of view. But, of course these types of applications are limited to certain functions. Automatically generated applications can contain only those functions that have

the appropriate implementation or support in the mobile platform-specific libraries, in the supporting SDKs or APIs. In contrary, the goal of our solution is to speed up the development and not to eliminate the native programming. We use software modeling to design different aspects of the mobile applications and generate some part of the source code for different mobile platforms. Our approach supports and often requires further development activities after the source code generation, e.g., to integrate the generated source code into the already existing source code, or to extend the functionalities with platform-specific native code. We still believe that each software application requires some human contribution on the programming level. The goal is to cut down the required time to complete the tasks, to effectively support development efforts, but not to fully eliminate manual programming.

Further difference is that the presented multi-mobile platform solutions are providing hybrid applications, i.e. the applications are partly web applications and partly they are based on platform-specific libraries. Our solution produces truly native applications, therefore usually they are providing better performance, and usually it is easier to perform their testing and management.

The Vision Mobile Developer Economics study [33] states that it does not make sense for a startup to do native *development for multiple platforms, both in terms of time and money.* We agree, but if we provide the appropriate methods and tools, like the presented one, then the native development for multiple platforms can be available for small and medium sized companies as well.

Backend as a service (BaaS) [7] is a model for supporting mobile and web application developers with common functionalities as services. The services are provided via the use of custom software development kits (SDKs) and application programming interfaces (APIs). Providing a consistent way to manage backend data means that developers do not need to redevelop their own backend for each of the services that their applications need to access, potentially saving time and increasing the quality because reusing tested solutions.

Different BaaS solutions offer a slightly different set of backend services [28]. Among the most common services provided are user management, file storage and sharing, push notifications, integration with social networks, location services, messaging and chat functions, and running business logic. There are several BaaS providers, for example, Parse [25], cloudbase.io [6] and Buddy [5]. They mostly differ by the set of services provided, the platforms supported and the pricing, however, as they are constantly evolving, one can not make a general selection disregarding the actual development project.

The Parse cloud application platform supports iOS, Android, JavaScript, Windows 8, Windows Phone 8, and OS X platforms. Parse provides scalable backend solutions, push notifications, social integration, data storage, and custom logic possibility. cloudbase.io maintains and scales backend infrastructure including push notifications, database management and mobile analytics. Buddy provides the Buddy Development Platform and the Buddy Analytics Dashboard to support application development and service providing.

The method provided by this paper is not competing with BaaS solutions, but it is about to utilize them. Utilization means that some library functionalities are

realized with different BaaS services.

7 Conclusion

People use their mobile devices every day to access a wide variety of digital content. We have seen that the diversity of mobile platforms and that of mobile device capabilities requires providing applications for each different platform. The paper has provided a technology for developing multi-platform mobile applications. We have also introduced our model-driven solution for developing mobile applications for multiple mobile platforms. This approach increases both the efficiency of mobile application development and the quality of the resulting software artifacts. This is achieved by providing a mobile, platform-independent, high-abstraction level environment for mobile application design. We support it with innovative, mobile domain-specific languages and effective model processing solution.

We work on to support mobile application developers, i.e. we continuously extend the capabilities of the introduced approach and framework by covering more areas of mobile applications.

References

[1] Aczél, K. and Charaf, H. Automatic user interface code generation in Symbian, MicroCAD 2005, International Scientific Conference, Hungary, pages 1–5, 2005.

[2] Appcelerator platform homepage, *http://www.appcelerator.com*

[3] Android webpage, *http://www.android.com/*

[4] Angyal, L., Asztalos, M., Lengyel, L., Levendovszky, T., Madari, I., Mezei, G., Mszros, T., Siroki, L. and Vajk, T., Towards a fast, efficient and customizable domain-specific modeling framework, In Proceedings of the IASTED International Conference, pages 11–16, Innsbruck, 2009.

[5] Buddy, *http://www.buddy.com/*

[6] cloudbase.io, *http://cloudbase.io/*

[7] DZone's Definitive Guide to Cloud Providers, *http://www.dzone.com/page/comparison-guide-to-cloud-providers-2013*

[8] Fielding, R.T. and Taylor, R.N. Principled design of the mod-ern Web architecture, *ACM Trans. Internet Technol.* 2(2):115–150, 2002. DOI=10.1145/514183.514185 http://doi.acm.org/10.1145/514183.514185

[9] Firefox OS, *http://www.mozilla.org/en-US/firefox/os/*

[10] Foley, M.J. Microsoft's Windows Phone 8 finally gets a 'real' Windows core, *http://www.zdnet.com/blog/microsoft/microsofts-windows-phone-8-finally-gets-a-real-windows-core/12975*

[11] Fowler, M., Domain-specific languages, Addison-Wesley Professional, 2010.

[12] Gartner survey 2010, *http://www.gartner.com/it/page.jsp?id=1529214*

[13] The Gelmato Netsize Guide 2013: M-commerce on the move, 2013.

[14] Harrison, R. Symbian OS C++ for mobile phones: Program-ming with extended functionality and advanced features, John Wiley & Sons, ISBN 0470856114, 2004.

[15] iOS, *http://www.apple.com/ios/*

[16] Li, S. and Knudsen, J. Beginning J2ME: From Novice to Professional, Apress, ISBN 1-59059-479-7, page 480, 2005.

[17] Lengyel L., Levendovszky T. and Charaf H. Applying Multi-Paradigm Modeling to Multi-Platform Mobile Development, In Proceedings of the Workshop on Multi-Paradigm Modeling: Concepts and Tools. Nashville, USA, 2007.09.30-2007.10.05. pages 9–21.

[18] Lengyel L., Levendovszky T., Mezei G., Forstner B. and Charaf H. Towards a model-based unification of mobile platforms, In 4th IEEE/ACS International Conference on Computer Systems and Applications, Sharjah, 2006. IEEE, pages 866–873.

[19] Levendovszky T. Lengyel L. Mezei G. and Mszros T. Introducing the VMTS mobile toolkit, 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007, Kassel, Germany, 2007, pages 587–592.

[20] Madari I. and Lengyel L. Synchronizing user interfaces of different mobile platforms, International IEEE Conference Devoted to the 150-Anniversary of Alexander S Popov: EURO-CON 2009, Saint-Petersburg, Russia, 2009. New York: IEEE, pages 1852–1859. ISBN: 978-1-4244-3967-6.

[21] Mono Project homepage, *http://www.mono-project.com*

[22] OMG Model-Driven Architecture (MDA) specication, OMG document ormsc/01-07-01, *http://www.omg.org/*

[23] OMG UML specification, version 2.3, OMG document for-mal/2010-05-03, *http://www.uml.org/*

[24] Microsoft's Text Template Transformation Toolkit (T4), Code Generation and T4 Text Templates, *http://msdn.microsoft.com/en-us/library/vstudio/bb126445.aspx*

[25] Parse, *https://parse.com/*

[26] PhoneGap homepage, *http://phonegap.com/*

[27] Richardson, L. and Sam, R. RESTful web service, O'Reilly Media, 2007.

[28] Rowinski, D. The Rise of Mobile Cloud Services: BaaS Startups Grow Up, *http://readwrite.com/2012/04/17/mobile-backend-as-a-service-ec#awesm= oiUfyQ4lngjNcS*

[29] Rubino, D. Overview and review of Windows Phone 8, 2012, *http://www.wpcentral.com/overview-and-review-windows-phone-8*

[30] SOAP Version 1.2, *http://www.w3.org/TR/soap/*

[31] Thai, T. and Lam, H. .NET framework essentials, O'Reilly, 2003.

[32] Vision Mobile: Developer Economics 2013, *http://www.visionmobile.com/devecon.php*

[33] Vision Mobile, Developer Economics Q3 2013: State of the Developer Nation, 2013.

[34] Vision Mobile, The European App Economy, 2013, Creating Jobs and driving growth, 2013.

[35] Visual Modeling and Transformation System, *http://www.aut.bme.hu/vmts*

[36] Xamarin homepage, *http://xamarin.com/*

[37] Yao, P. and Durant, D. .NET compact framework program-ming with C#, Addison-Wesley Professional, 2004.

Received 13th January 2014

Acta Cybernetica **21** (2014) 553–583.

Versatile Form Validation using jSRML

Miklós Kálmán *

Abstract

Over the years the Internet has spread to most areas of our lives ranging from reading news, ordering food, streaming music, playing games all the way to handling our finances online. With this rapid expansion came an increased need to ensure that the data being transmitted is valid. Validity is important not just to avoid data corruption but also to prevent possible security breaches. Whenever a user wants to interact with a website where information needs to be shared they usually fill out forms and submit them for server-side processing. Web forms are very prone to input errors, external exploits like SQL injection attacks, automated bot submissions and several other security circumvention attempts. We will demonstrate our jSRML metalanguage which provides a way to define more comprehensive and non-obtrusive validation rules for forms. We used jQuery to allow asynchronous AJAX validation without posting the page to provide a seamless experience for the user. Our approach also allows rules to be defined to correct mistakes in user input aside from performing validation making it a valuable asset in the space of form validation. We have created a system called jSRMLTool which can perform hybrid validation methods as well as propose jSRML validation rules using machine learning.

Introduction

Information exchange has become a vital part of our lives. The Internet is the key channel to provide the means to digitally exchange data between its users. The number of users hooked up to the Internet is increasing day by day. Social networking sites engulf the ether and integrate with our lives. With this growth comes an ever-increasing amount of data being transmitted. Users perform their daily tasks online, giving out information, submitting data on sites. Data integrity and security is a vital concept in this eco-system. The most common form of user initiated information exchange are web pages. These pages are written in HTML[1] and may contain web forms that consist of fields. These fields are filled out by the user, which are then submitted to the server for processing. The server then processes this information and returns the results or performs an operation with the

*University of Szeged, Department of Software Engineering, Dugonics tér 13., H-6720 Szeged, Hungary, +36 70 3684910, email: mkalman@inf.u-szeged.hu

submitted data. These web forms can range from simple user login forms all the way to online tax returns containing and exchanging sensitive information. Unfortunately this is one of the weakest links in the whole system as many hackers try to exploit sites through their forms. The most common form of attacks against web forms is *DoS*[2] (Denial of Service), which basically means that small automated scripts perform constant form posting against sites trying to exploit the data or cause the service to slow down or even crash. This can potentially compromise the site granting the malicious script access to protected resources. This type of exploit is also used to spam forums and news portals. Even if the data transmission itself is protected using a secure channel (e.g.: SSL) the data entered still needs to be validated prior to performing the processing. Another common exploit method is the notorious *SQL injection attack*[3]. This method is based on the assumption that the fields of the forms are eventually inserted into the database. If the form processor does not filter the input (e.g.: by using prepared statements, or by filtering the fields for SQL commands) then it is very possible to issue SQL commands against the processing database (for example DROP TABLE). Aside from a security point, data validity is a crucial aspect as well. Consider a lead generation form where users need to fill in their contact information in order to receive special offers from the provider. If the data entered is incorrect then it can cause a potential lead to be lost causing the owner monetary damage.

One of the most common types of validation scenarios is the user registration form. Here the user fills in his personal information, along with an email and password and submits it for processing. The email address has to be valid, otherwise the provider cannot communicate with the user, the passwords have to conform to some security restrictions...etc. All these requirements can be handled by using some kind of form validation method. The most common is asynchronous validation using *JavaScript*[4]. Using this approach the author of the page writes JavaScript code which checks the fields of the form providing visual output to the user (e.g.: if the email has an invalid format then the field may be highlighted). This type of validation can be very powerful and is handled on the client side, which means the user will not experience any lag during the submission. The biggest drawback however is that by adding more fields to the form the JavaScript code processing logic becomes more difficult.

The second type of form validation is *Server-side* validation. This basically means that the form data is posted to the server, which then processes the content and returns an error if the form was invalid, or saves the data if it was valid. This is a good approach, however it will cause an overhead when the user has to re-enter the form contents due to a mistype in one of the fields unless the owner explicitly codes the retry logic. The process will not happen asynchronously, meaning the page will be reloaded during the submission (excluding cases when this is handled with an AJAX[5] call).

To provide a solution to these issues we have created a jQuery[6] based validator called jSRMLTool which leverages the SRML[7] language we introduced in one of our earlier articles. This language was extended to allow form based validation rules. The original SRML specification targeted XML document compaction and

decompaction. With our new jSRML extension users will be able to define SRML rules for web forms and their fields, describe relationships and requirements for their content. The engine can be used in any HTML page simply by including the script file in the document and defining the validation rules. This approach ensures that the HTML content is not encumbered with JavaScript code. The jSRML rules need to be placed after each field that is to be validated and the engine will handle the rest. We will detail how this approach works in a later chapter of this article.

An off-site asynchronous implementation of the jSRML engine was also created using Servlets capable of validating forms using unique identifiers and jSRML rules. This is a separate service running on a remote machine using stored rules to validate the form and return with any potential validation errors. Our approach also allows another powerful feature: data correction. Thanks to the nature of the jSRML language, it is possible to define self-correcting form validation rules. These rules correct the field values based on the rule definitions wherever applicable making the form submission succeed. The Servlet also has provision to learn potential jSRML rules using the submitted form data and machine learning.

We will start out by providing some basic background on the technologies used throughout the article. We then continue on to show the extension made to the SRML language that allow for the definition of form validation rules. Afterwards we will demonstrate the potential of learning jSRML rules using the jSRMLTool servlet and evaluate the results. We end the article by an analysis of related work in this field finishing off with a summary and our plans for future expansion.

1 Preliminaries

Before we introduce our new method we should cover a few topics in order to make the article easier to understand. We will not detail each technology too much, rather just cover the parts that are relevant to the later sections.

1.1 HTML and DOM

Forms are described using the HTML[1] language. These documents have a similar hierarchic structure to XML where each node can contain attributes or additional child nodes. This hierarchic tree-like representation is also known as the *DOM model*[8] (Document Object Model). *Figure 1* shows a simple HTML form source with a field. The DOM tree representation of *Figure 1* is shown in *Figure 2.*

1.2 Types of form validation

There are four major types of form validation: *Client-side*, *Server-side*, *Real-time* and *Hybrid.* The difference between them lies where the data is validated and processed.The different types of form validation are summarized in *Figure 3.*

```
<html>
   <head><title>Hello World</title></head>
   <body>
     <h1>Hello World!</h1>
     <form method="post" action="process.php">
       <label for="username">Name:</label><input type="text" name="username" />
       <input type="submit" value="Submit" />
     </form>
   </body>
</html>
```

Figure 1: Simple HTML of form

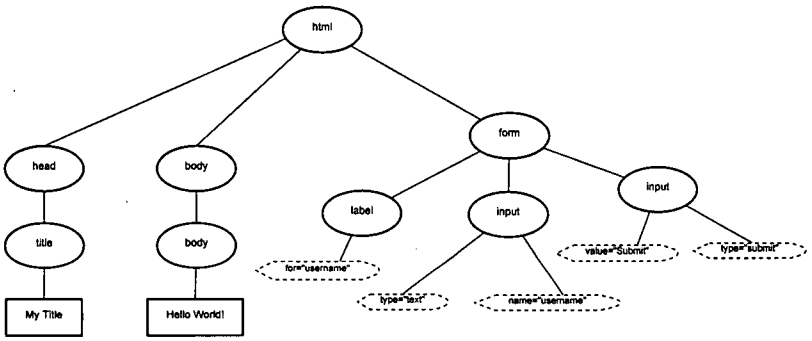


Figure 2: DOM tree of the Form Example

Type	Trigger	Processing	Validation logic	Advantage	Disadvantage
Server Side	Form Submit	Sequential	Returned to browser for display of results	Validation logic hidden from user	Validation changes require server updates
Client Side	OnClick intercept	Client side	Shown in browser using JavaScript	Fast since no data is sent to server	Validation logic visible to users
Real	Field change	Either	Direct call to client and/or Server validation	Field values validated realtime prior to form submission	More traffic required, harder to update
Hybrid	Field change and Submit	Either	Direct calls with roundtrip to server	Allows two stage validation, pre-filtering results prior to sending to server	More complex to implement and maintain

Figure 3: Validation types

1.3 SRML

The SRML[9] metalanguage was introduced to allow the description of semantic rules that can be used to compact and decompact XML[10] documents. The term

compaction comes from the fact that it is able to remove specific attributes based on rules and can recreate the same value (therefore restoring) at any later time. The original SRML rule engine implementation used the DOM tree of the XML to perform its operations. Since HTML forms can be considered as DOM[8] trees it made sense to attempt to apply SRML to this area as well. In this article we introduce an extension of SRML (called jSRML) which allows its use in the form validation space. We have created a new rule engine for this purpose using jQuery where the processing is performed in the browser.

The new jSRML language although being an extension of SRML is not completely similar to its predecessor as it was rebuilt from ground up taking the positive traits of the previous language version and molding it to become an ideal candidate for describing form validation rules. *Figure 4* shows the differences between the different versions of SRML.

Property	SRML 1.0	jSRML
Main Focus	Compaction	Validation/Correction
Reference level	Attributes	Form Field values
Application Area	XML Documents	HTML Forms
Rules based on	Attribute Grammars	XPath and DOM
Rule Definition	Complex	Simplified
Rule Locations	DTD and SRML file	Inline, external, server
Rule Processing	Application side	Client-,Server-side, Mixed

Figure 4: Key differences between SRML versions

2 Extending SRML for form validation

In this section we will present how the SRML language can be extended to aid the validation process. Most *Client-side* validators are simplistic and perform format validation only. If we wanted to create a validation rule that conditionally compared two fields then it would require a larger block of JavaScript. Trying to achieve this on the server would require the validation logic to be implemented there. If for some reason the conditions needed to change then the server code would need to be updated, which can be difficult in production environments.

We took the positive traits of the original SRML engine and rebuilt it from the ground up in JavaScript using jQuery to allow exceptional browser performance. We decided to name the extension jSRML and the new rule engine jSRMLTool to denote the JavaScript relationship. Previously SRML rules were stored in a separate file which had its advantages and disadvantages. The advantage was that all the rules were in one location, however this also meant that it was harder to understand the rules when trying to find a ruleset for a given node context. In the jSRML approach we allow the rules to be defined in-line after each field as well as externally making it easier to define validation rules.

The second advantage of jSRML is that it is non-obtrusive. In order to use it only a simple script include is required. When the validation rules need to be

updated the rule engine itself will not change, only the rules, reducing the possibility of error. This is a very large benefit compared to the pure JavaScript approaches. If the validation rules need to change then only the affected field rules need to change, no coding experience is needed to perform the update. In case of in-line jSRML, the rules are defined as jSRML snippets. The full XSD of the new jSRML language can be found in [11].

The jSRML engine can also correct the field values if the rule definition specifies it. This is a huge advantage over other rule- or JavaScript-based validators as it allows the form to correct the errors and still allows the form submission to succeed. A good example would be spell checking in a form prior to submission which can be accomplished by the using functions in the rule definition. This makes jSRML more versatile as more seasoned developers can extend the engine with additional methods aside from the standard operation set that the engine provides.

We have also created a *Server-side* implementation of the jSRML engine using Java Servlets[12] allowing the form to be validated asynchronously against a service. The service code does not change no matter what the rule definitions are. This is accomplished by storing the ruleset on the server-side and performing the validation based on a lookup using a unique form identifier. This Servlet can be used to validate thousands of different forms spanning multiple domains as long as the rules were uploaded beforehand. This allows the engine to be leveraged in an on-demand validation service scenario. The jSRMLTool servlet also has an option to learn the validation rules based on the form inputs using extendable machine learning methods. This provides a powerful tool for the owner as it can also "mine" the input and gradually adjust the rules based on what users entered.

3 Validation using jSRML

We will show how to define jSRML rules using simple snippets. The current language format allows two ways of defining rules : *in-line* and *external*. The *in-line* mode allows the user to insert the validation rules right below the affected field. This makes the code more readable as the validation rule follows the field itself. *Figure 5* shows a simple example of providing an email validation rule using *in-line* jSRML.

To initialize the engine for in-line (default) validation mode the following steps would be needed:

- Include the *jSRMLTool.js* file at the start of the document.
- Augment the fields with their proper in-line rules.

In-line validation rules are contained in a comment block following the field. The comment starts with the [SRML] tag. The advantage of using comments for the rule storage is that they are non-obtrusive and can be accessed within the DOM model using XPath expressions. XPath[13] is a query language allowing the easy access and manipulation of nodes and their content within a DOM tree.

```
...
 <input type="text" id="email"  class="row-item" />
   <!--[SRML]
     <validate-input id="email" form="myform" mode="validate">
        <error-text>Invalid email format!</error-text>
        <css invalid="inp-form-error" error-class="form_error_message error" />
        <action valid="" invalid="error" />
        <conditions>
            <expr>
                <text-format value="email" />
            </expr>
    </conditions>
     </validate-input>
-->
...
```

Figure 5: jSRML snippet for in-line email validation

For external includes we use jQuery to load an XML document containing the rules into a DOM object and use that as the source for the engine. As this is not the default mode that the engine uses there is some extra setup required for this mode to be used. To use external rules the following steps need to be taken:

- Create a script segment with the following contents :

```
var external_rule = http://location-of-srml-rules;
```

- Include the *jSRMLTool.js* file.

The major difference between *external* and *in-line* is that there is an extra step required. The presence of an *external_rule* variable informs the jSRMLTool engine to load the rules from that location using AJAX during the page load. The rules are then pushed into a rule DOM object for easier access. From this point on the validation process is identical to the *in-line* approach.

3.1 Defining validation rules

After demonstrating the two ways to define rules we will now describe how a rule is built up and how to define more complex ones.

Every jSRML rule definition starts with the `validate-input` tag. This element specifies what the scope of the given rule is using the *id* attribute. The *form* attribute defines which form the rules belong to. This way the *external* and *in-line* rules can both use the same format making it easy to switch between them. The third parameter is the *mode*, which can have a value of *"validate"* or *"correct"*. The first mode will validate the rule and return accordingly. The *"correct"* mode allows the form input field to be corrected by the actual rule calculation result. This means that if the validation fails, then the field value will be replaced by a predefined or calculated value (Expected value) allowing the validation to potentially finish successfully.

The `validate-input` element has 4 child nodes. These can be in any order, but they must exist for the validation to yield proper results. These elements are as follows:

- **error-text:** This element contains the validation message that will be displayed to the user. This message is put in a dynamic *div* element that is created after the field that is being validated. A *div* is an HTML element which can have an *id*, *name* and *class* attribute. Divs are used in modern web pages to provide table-less layouts and define specific regions of the page. For the scope of this article it is enough to consider them as containers that can be manipulated similarly to other DOM elements.

- **css:** The css element allows the author to define what CSS classes should be amended to the input field in case of an error and what class the newly created error div should be. CSS[14] stands for Cascading Style Sheets and is widely used in styling web pages. It defines a set of styles and classes which can be applied to elements in the document.

- **action:** This element allows the definition of additional functions that will be invoked in case of a validation error or success. This allows more extensive callbacks to experienced users who wish to perform custom operations depending on the output of the form validation results.

- **conditions:** This element stores all of the validation rules.

The `condition` tag contains one or more `expr` tags. The validation succeeds or fails based on the result of these expressions. It is possible to define more conditions for the same field using multiple `expr` nodes. There are several expression types defined in jSRML. We will detail the most important ones along with a brief description.

- **binary-op:** This defines a binary operation. In jSRML we only allow a subset of `binary-op` types on the top level expression, more specifically ones that return a true/false value. Currently these are limited to: *gte*, *gt*, *lte*, *lt*, *date-lte*, *date-lt*, *date-equals*, *date-gt*, *date-gte*, *equals*, *not-equals*, *contains*, *not-contains*,*begins-with* and *ends-with*. The specification also allows the keywords *and* and *or* to enable proper logical operations. We have introduced the *reg-eval* element which allows references to nodes and most binary operations (+, -, /, *). A `binary-op` contains two `expr` expressions. The operation is performed between the two expressions. The expressions within can also be other binary-ops or one of the expression types described in this chapter.

- **text-length:** The `text-length` element returns the length of the actual field that the rule is defined for.

- **field-length:** This element is similar to `text-length` however it also has an attribute called *id* that identifies the specified field whose length needs to be returned.

- **text-value:** This expression will return the value of the actual field that the rule's definition was for.

- **field-value:** Similar to `text-value` but allows the reference of another field's value by *id*.
- **data:** The `data` element allows literals or constants to take part in an expression. An example for this would be when the length of a field has to be larger than 100. In this case the 100 would be added as a `data` tag.
- **text-format:** The `text-format` expression returns true or false based on the type of field value it is matched against. The *value* attribute can be *date*, *numeric*, *email* or *regexp*. This allows easier validation against standard field types used in forms, like emails, dates or numbers. The *regexp* type allows the definition of a regular expression defined in the *expression* attribute. This allows powerful pattern matching for fields (e.g ISBN number validation).
- **reg-eval:** This expression type allows operations to be defined on more fields at the same time. For example if the field value is only valid if it is the sum of other two fields then a `reg-eval` expression can be used. To reference the value of fields in the expression one simply needs to enclose the *id* of the fields in brackets (e.g.: $[\{fieldName\}]$).
- **if-expr:** The `if-expr` element allows conditional results to be returned. It takes 3 `expr` expressions. If the result value of the first expression is true then the result of the `if-expr` will be that of the second `expr` otherwise it will be the third `expr`.
- **has-value:** This element allows a simple check of the field contents. If the field referenced by *id* is empty this element will return false, otherwise it will return true.

The jSRML language allows the form values to be corrected based on the rules. The engine will find the rules for the actual field and if the value of the field is different than the expected value defined then it will use the result of the rule as the actual value. This allows forms to be corrected based on the rule values making it a very powerful tool in the form validation space.

3.2 A form validation example

After introducing the jSRML language and how powerful it can be for form validation we will provide a summary example to demonstrate how it can be used for form validation.

Consider the form in *Figure 6.* This form has multiple fields to better demonstrate how jSRMLTool works. The full source of the page can be found in [15]. The following shows some summarized validation rules for the form:

- `Field01 has a minimum length of 5 characters`: the `text-length` element is used which returns the length of the actual field (in this case the length of *field01*). We then compare this to a constant value of *5* defined

in a `data` element. To perform the comparison logical operator we use a *gte* binary op. This will return true if the first expression's value is larger than the second.

- `Field04 has to be an ISBN number:` This is a special `text-format` case as it is using the `reg-exp` type to define a requirement of an ISBN number. The *expression* attribute defines the actual regular expression that the field's value will be validated against.

- `Field06 has to be the sum of Field02 and Field05:` For this rule we use `reg-eval` which is coupled with an *"equals"* `binary-op` against the actual text value.

- `Field11 is ''legs'' if field10 is ''cat'', ''wings'' if field10 has a value of to ''bird'' and can be anything otherwise :` The validation rule contains an `if-expr` to match the value of the other field value against *"cat"*. If the value was *"cat"* then the validation result will return the value *"legs"* as the required field value. Otherwise the results will be the `text-value` of the node and will perform an *"equals"* `binary-op` on it. This is a simple trick to convert the machining of fields to booleans, since if the value matched then we return the current field value and compare that against itself (which will always be true), otherwise we would return *"legs"*.

The jSRMLTool engine supports all three types of validation described earlier (*Client*, *Server*, *Real-time*). This provides the most versatile and powerful approach since the user is not bound to a single solution.

The following summarizes how the different modes operated in jSRMLTool:

- `Client-side:` In this mode the validation is completed using the included jSRMLTool.js file. The rules are extracted using XPath conditions. All *inline* rules are contained in comments which start with [SRML]. A hook is installed on the *onClick* action of the submit button. When the button is pressed the engine will validate the fields. If the validation is successful (or corrected based on the expected values) then the form is submitted to its original location defined by the "action" attribute of the form. *Figure 7* shows the flow of the *Client-side* validation.

- `Server-side:` The engine handles the *Server-side* mode using a separate servlet (called jSRMLToolServlet). This servlet uses a unique identifier to associate the rules to each form. This allows multiple forms from different domains to be submitted/validated against the same servlet. To put the validation engine into server mode a variable called *server_validator* needs to be defined with the URL of the servlet. The flow in this case is similar to the *Client-side* however all fields are pushed over to the servlet along with the unique identifier. The servlet then performs the validation/correction and returns the data back to the client. The *Server-side* validation flow is shown in *Figure 8*.

- `Real-time and Hybrid:` Every rule has a *"method"* attribute. This is not a mandatory attribute and has a default value of *"standard"*. When this attribute is set to *"focus"* then a hook is automatically installed on the *onBlur* event of every field where this attribute is set. This results in a focus change validation trigger. The third allowed value for the *method* attribute is *"real-time"*. This installs a *keydown* listener and performs the validation on every character input. This mode is useful for example in case of password length checks.

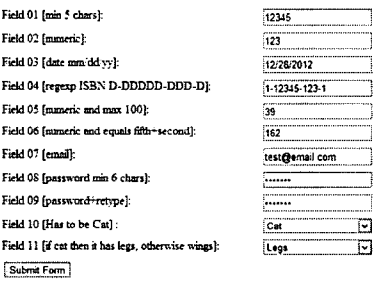


Figure 6: Input form

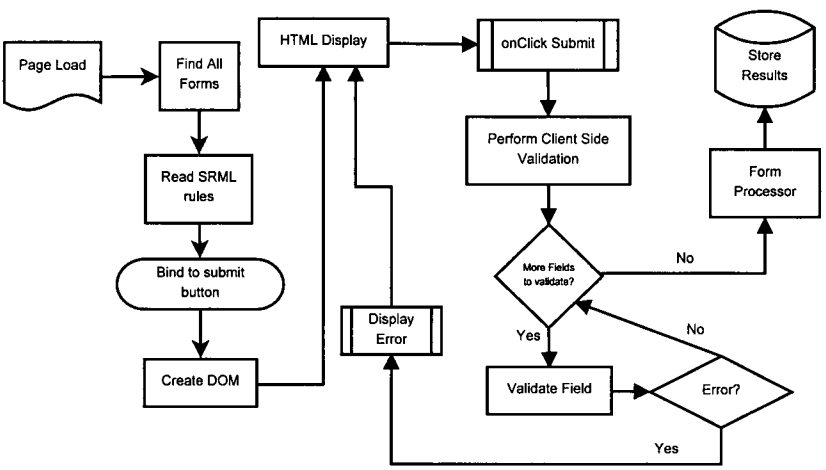


Figure 7: Client-Side jSRML

4 The jSRMLTool Servlet

After introducing the jSRML language and the jSRMLTool engine we will now discuss the *Server-side* validation mode in more detail. The jSRMLTool servlet

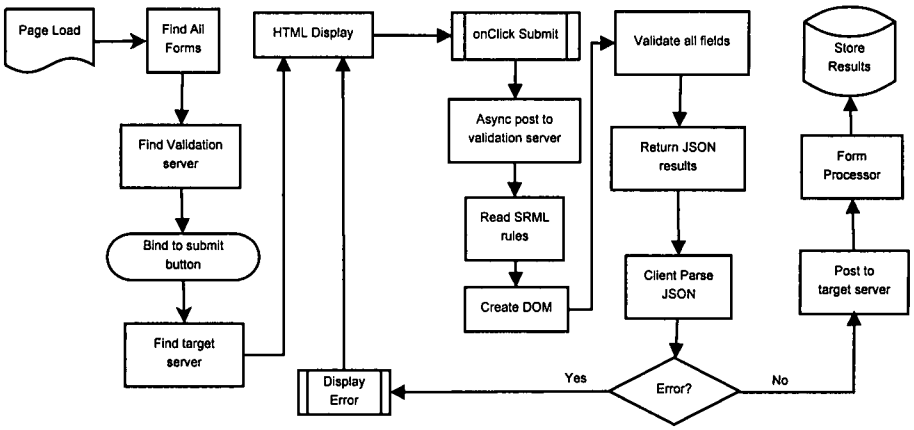


Figure 8: Server Side jSRML

has two major roles: *Server-side* form validation and learning jSRML rules. The first role allows a powerful way to provide a service for validating forms across multiple servers. The jSRML rules are stored in the database and are retrieved using unique identifiers. The form is passed in to the Servlet which performs the validation internally and returns the results to the calling client. This approach hides the rules from the client side, yet still allows powerful validation using jSRML.

4.1 Learning jSRML rules

The second role of the jSRMLTool engine is learning jSRML rules. This is a powerful addition since it attempts to learn from the form submissions and can propose jSRML rules based on machine learning techniques. In order to learn jSRML rules, the engine has to be put into learning mode using the following steps:

1. Create a JavaScript variable called *server_mode* with a value of *"learn"*. This will put the engine into learning mode. The default value of this variable is *"normal"* .

2. Create a variable called *server_validator* with the location of the validation servlet.

3. Include the jSRMLTool.js file into the header of the form's file similarly to the *client* or *server-side* modes.

4. Augment the form with a hidden variable called *srml_unique*. The value of the variable should be the identifier that will be used to group the form submissions together.

Figure 9 demonstrates how the form is intercepted and analyzed. The initial steps are similar to how the *Server-side* validation is handled. A hook will be

installed on the form's submit event and will re-route the call to the jSRML Servlet location. The major difference here is that there is no actual jSRML ruleset on the Server-side. It is merely used to intercept any submissions and store the form-value pairs. These values are then analyzed by the learning module and possible jSRML rules are generated. The flow is returned to the client and the form data is pushed to the original target for the form submission. This means that the form operation is not hindered but the traffic is intercepted, saved and submission relayed to its original target.

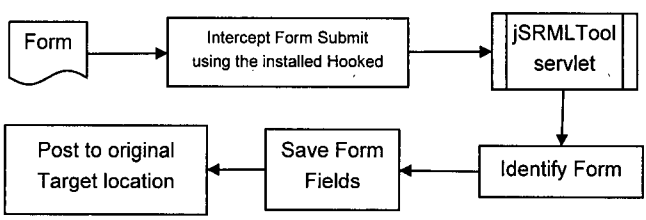


Figure 9: Intercepting form data and learning jSRML rules

The learning module has several plugins that process form submissions and adjust the proposed rules accordingly making the learning a gradual process. Currently the engine has the following learning plugins: *jpFormat*, *jpLength*, *jpCopyContent*, *jpRelationship*, *jpRange*, *jpPredefinedName*, *jpRegExp*. We will detail each learning plugin in this section.

Each plugin has a *confidence factor* and a *target ratio* that is set by the administrator of the system. If a plugin has a high *confidence value* it means that almost every time the plugin breaches the target ratio threshold a rule will be generated. Sometimes it is possible that multiple plugins provide rules for the same field. In cases like this the system chooses the solution with the highest *confidence factor* which surpassed the *target ratio*. The *target ratio* denotes what the minimum expected matching ratio is, which means that if the actual match is lower than this ratio the rule will not be considered as a match. In practice this means the ratio of inputs that match the given rule conditions.

The plugins keep track of their historical form submissions along with their field values. The learning module goes through all the plugins and collects the partial jSRML rule proposals. Once all the plugins are executed the weighed results are analyzed and stored. *Figure 10* demonstrates how the learning module works. To increase the efficiency of the learning process it is usually helpful to start a new ruleset with a supervised learning scenario. During this the owner of the form "teaches" the engine by providing valid sample inputs. Sometimes previous valid form submissions are also available in bulk. The tool also has an import feature which is able to import a CSV file of valid sample data to prime the initial rules. Since the learning module is very extensible, new plugins can be added easily. This can increase the learning efficiency of the system.

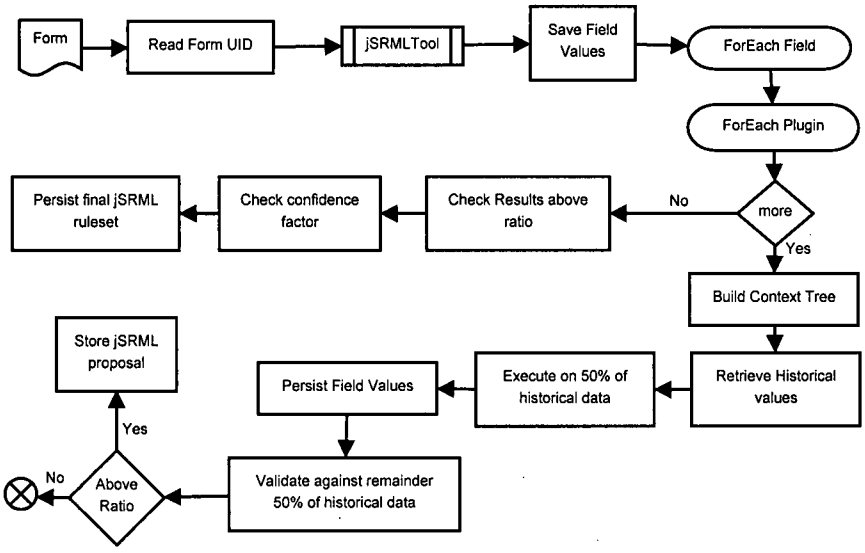


Figure 10: jSRMLTool learning process

4.1.1 jpFormat Plugin

This plugin tries to match the type of a given field. It works on a simple approach that every field is a *string* as the weakest type match. It then tries to cast to *date*, *email* and *numeric*. The matching is done by casting and regular expression pattern matching. The results are stored on a fieldname level along with the statistics of the match. The decision adopts over time since it is possible that not all submissions are valid. The plugin has a high success rate at identifying the formats, since the more positive/negative examples it receives the higher probability the match will be.

4.1.2 jpLength and jpRange Plugins

The *jpLength* plugin matches on the length of the fields. Both minimum and maximum lengths are collected and analyzed. The operation is pretty straightforward thanks to the historical data collected. The *jpRange* plugin works similarly, however with the actual numerical value of the fields. The range, min and max values are adjusted after each positive result. These plugins are dynamic in nature and adjust their values based on the submissions.

4.1.3 jpCopyContent

This plugin is a simple comparator between two fields. It is mostly used in the password, email fields when there is a second field which requires the user to retype the value to ensure he didn't make a mistake. The operation of this plugin

goes through all (F_j, F_k) field pairs and checks what the matching ratio is between them.

4.1.4 jpRelationship

The relationship plugin is aimed at finding relationships between fields and their values. The steps of the plugin are demonstrated in *Figure 11.* The learning starts out by extracting the context of the form submissions. Since the context tree has only two levels (including the root) every field is a sibling. This plugin has two sub-modes: *compositional* and *conditional.*

The *compositional* mode finds potential compositions between the other sibling elements. The current version works off sets of two concurrent fields at a time (using more fields would increase the complexity), each field with a minimum length of 3. Based on the possible combinations we build a statistical table to show each field in relation to two other siblings. For composition we check against: `begins-with`, `ends-with`, `contains`. If *field01* is the field the plugin is targeting and *field02* and *field03* are in the current context set then the value is compared against: *[field02][field03]*, *[field03][field02]*, **[field02]*, **[field03]*, *[field02]*[field03]*, *[field03]*[field02].* The plugin will go through every field as the target field. It then takes the remainder *(n-1)* siblings and splits them into groups of two based on those fields whose lengths are above 3 characters. These combinations are then compared to the historical values of the plugin. Based on the *confidence factor* and ratio provided a jSRML rule is created. *Figure 12* shows the compositional method of the plugin.

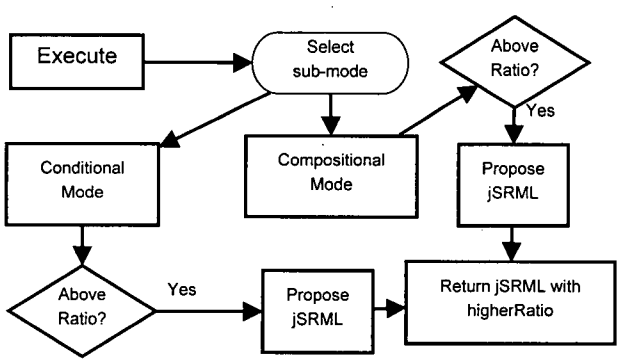


Figure 11: jpRelationship Plugin

The second mode of the *jpRelationship* plugin is the *conditional* mode (*Figure 14*). This method finds relationships between field values using conditional logic and applying statistical machine learning[16]. The plugin uses 50 percent of all historical data as the learning set. The plugin initially selects the most descriptive field F_k where $k=1,...,n$ and bags its context (the remainder *n-1* fields) clustering them into groups of three randomly. These clusters will form a set of decision trees

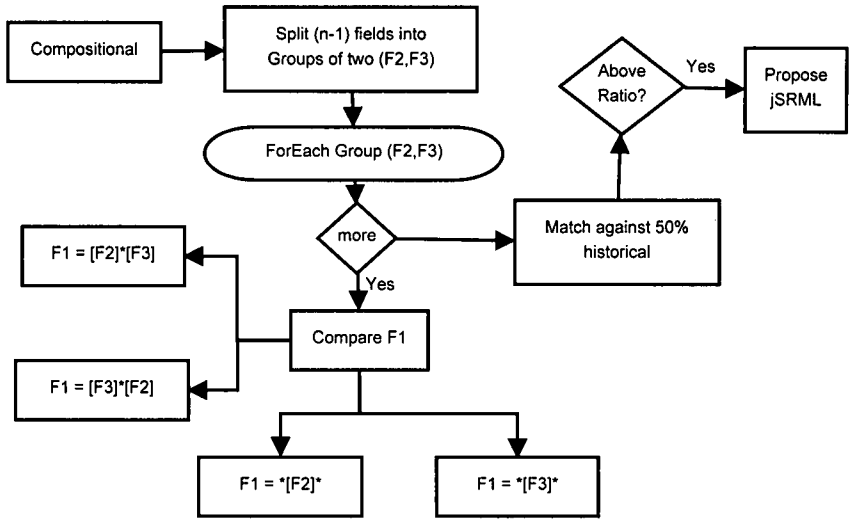


Figure 12: jpRelationship Compositional Method

that are focused on learning F_k using a simplified Random Forest[17] approach. It should be noted that the size of the clusters is an experimental value based on the average number of form fields per submission. The term "most descriptive field" refers to the field with the lowest entropy in the results (the field whose values are least random across submissions). This is used to better split the values of the results into smaller chunks which are then used in the later nodes of the tree. Every tree will have a maximum depth of 3 (as the selected field's bag has 3 other fields that have to be analyzed). Each node's content contains the actual values of targeted field F_k and its top three values (F_k was selected at the start of the algorithm). Every node will select the most descriptive field and its value in the current context. The context is unique to each node and the path that it was created by. This means that every field's possible values in the current node are influenced by the previously selected classifiers leading to the node. We will be using X_i to denote the filter context of a node in each iteration step whose value is unique to the node's path in the tree. Let $X_i := F_k[F_r = V_s(F_m[X_{i-1}]$ where $V_r(F_s[X_i])$ denotes the rth most descriptive value of field F_s filtered by the context defined in X_i. Let $C(F_r[X_i])$ mark the classifier that is selected for field F_r whose values are filtered by the context defined in X_i. During each node the field (F_r) with the most descriptive trait is selected as the classifier (every level of the tree reduces the number of fields to chose from by one). This field's values are then used to create the nodes children ordered by their descriptiveness. Each child node will fix the value of F_r based on the branch they are in $V_1(F_r[X_i]), ..., V_n(F_r[X_i])$. The main F_k field values and their occurrences are recalculated based on the context in each node. Every node will reduce the possible values of the fields as the context is generalized more going downward in the tree. It is possible that some field

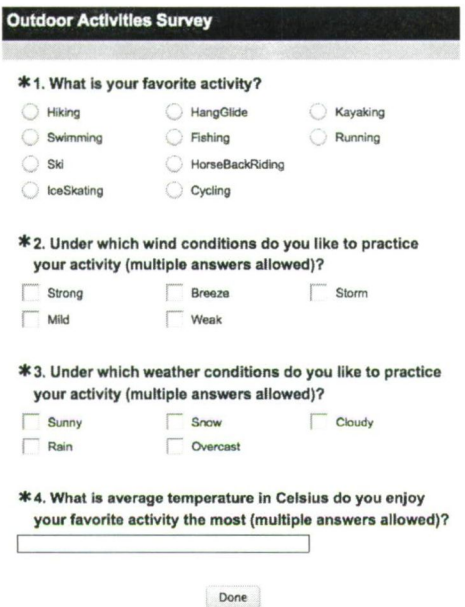


Figure 13: Outdoor Activities Form

values are not discrete, but rather continuous numerical occurrences. To solve this scenario $W_m(F_s[X_i])$ marks the weighed values of F_s filtered by X_i with a relation of m (possible values $\leq$, $>$). The algorithm chooses a weighed average of numeric values (to ensure that they are not offset too much). For these classifiers the values will partition the results into two sets. The first branch will contain values less than or equal to the classifier value, the second branch will contain values larger than the value. This function is analogous to the $V_m(F_n[X_i])$ value and can be used in the classifier filtering accordingly, however here the value is not based on the level of descriptiveness but rather the weighed average of the field and its filter chain.

As mentioned earlier each node contains the top three values of the analyzed field (F_k) with their occurrence ratio. The possible values of the fields are influenced by the previously selected classifier values. Before selecting a new classifier the algorithm checks the values of F_k in the nodes. Any node which does not have at least one F_k value above the ratio (currently set to 50%) is ignored from then on and will no longer be processed. The iterations continue until the context bag is not empty or all nodes have terminated without a possible selection. The algorithm only works off the top three values of each field classifier which may cause an efficiency decrease overall, however based on the introduced ratio values the margin for extra error can be safely ignored.

To demonstrate the algorithm consider the following example: users answer a set of questions regarding their activities and weather conditions ($activity[F_1]$, $wind[F_2]$, $weather[F_3]$, $temperature[F_4]$ where the brackets contain the Field index).

The form data was acquired using an online survey using the help of *SurveyMonkey*[18]. The fields *wind* and *weather* allow multiple values to be selected (the form can be seen in *Figure 13*). When the user selects multiple values for these fields the form post is handled as multiple submissions to fit the model correctly. The plugin uses 50 percent of the historical data (in our case 2000 submissions) and analyses each field one-by-one. We will demonstrate the *activity* field relationship learning briefly. *Figure 15* shows the resulting tree for *activitity* (note we only have 4 fields in this form, so it will only need one tree per field, however the algorithm works on multiple trees as described earlier). The plugin collects the distinct historical values and their counts selecting the top 3 values. In case of *activity* these top 3 distinct values are *"Swimming"* with 610 hits, *"Fishing"* with 239 hits and *"IceSkating"* with 215 hits. The learning set in our example is made up of 2000 form submissions.

The plugin creates a statistical analysis of the other $(C(F_2), C(F_3), C(F_4))$ classifier values. In our example $wind[F_2]$ is chosen as it had the most descriptive classification (provides the largest separation of results). The top 3 $wind[F_2]$ values are selected and the resultset is filtered on that $(V_1(F_2), V_2(F_2), V_3(F_2))$. If there are numeric values (e.g.: temperature) then the weighed average value is taken as the classifier. This however will only classify into two sets so they are only used in later levels of the tree.

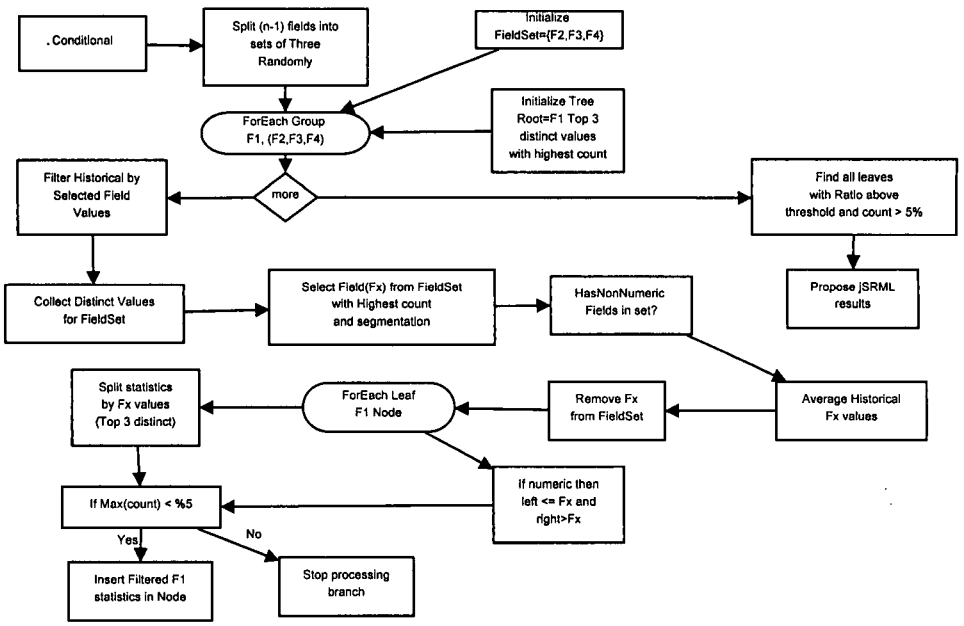


Figure 14: jpRelationship Conditional Method

The next tree level is created by applying a filter on the classifier results. In the example this means three nodes. The first node will list all entries where the *wind*

(F_2) is *"Weak"*, the second sibling will list all entries where the *wind* is *"Strong"* and the third node on this level will list all items whose *wind* attribute is *"Breeze"*.

Based on the new level we recalculate the top three distinct values of the target (F_1) field for each selected value of $V_i(F_2)$. On a database level this basically means that we `select the top 3 distinct values for` F_1 `where value of` F_2 `IN` $(V_1(F_2), V_2(F_2), V_3(F_2))$. The statistics are stored on the node level and are based on the filtered F_2 values.

The next step is to examine the remainder fields and create possible classifiers. The possible values of the fields are reduced by fixing field F_2 to the top three values. Based on the filtering weather (F_3) is chosen and the classifiers become: $C(F_3[F_2 = V_1(F_2)]), C(F_3[F_2 = V_2(F_2)])$ and $C(F_3[F_2 = V_3(F_2)])$ respectively. Taking the first classifier from the left the top three values it generates are *"Sunny"*, *"Rain"* and *"Snow"*. These values are used to filter all nodes on the level. On each level the distinct values of the F_1 are reduced based on the previous classifiers (e.g.: on this level only submission items that have the *weather* and *wind* values specified earlier are used to get the distinct values of the target F_1 field). The top three distinct values of the remainder two classifier are also generated and added to the tree.

The last level has only one field left to use: $temperature[F_4]$. Since this is a numeric value, we take the weighed average of historical values (taking into consideration the field values chosen for F_2 and F_3). Taking the left node as an example (the remainder nodes operate similarly) this classifier becomes $C(F_4[F_3 = V_1(F_3[F_2 = V_1(F_2)])])$. The left branch will be where the value of F_4 is less then or equal to the classifier's single value of *10* (weighed average of submissions for this field after applying the previous classifiers) and the right branch contains statistics on field values larger than this value. Once the tree is built we look at the leaf values. We select whichever ones breach the ratio provided (in our example we set this to be 50 percent). If more than one leaf on the same node breaches this threshold we select the largest one. If they are identical then we select the first one from the left. To avoid too many false positives we also have a concept of coverage ratio. This is set by default to 5 percent. What this entails is that all result counts below 5 percent of the learning dataset will be ignored. In the example this comes to 100 elements, which means that any leaf result below 100 submit matches are ignored. Based on our example the following jSRML rules are proposed:

1. *"Activity"* is "Swimming" (64 percent of the cases) when the *"wind"* is "Weak" and the *"weather"* is "Sunny" with a "temperature above 10 degrees"

2. *"Activity"* is "Swimming" (59 percent of the cases) when the *"wind"* is "Weak" and the *"weather"* is "Rainy" with a "temperature above 16 degrees"

Once a proposed prediction is made it is then checked against the remainder 50 percent of historical data to confirm that the matching ratio is kept. If the ratio is above the target ratio a rule is created. It is important to note that the validation ratio of this learning algorithm is not 100%. This requires the owner

of the domain or form to set the thresholds accordingly. It may mis-classify valid inputs as false negatives if the threshold is not set correctly. The purpose of the learning here is to provide a direction of validation rules that can then be refined by the domain owner in contrast to the other learning plugins which can classify the inputs with higher confidence. With more plugins and stronger learning algorithms (e.g.: neural networks) the system can evolve to better classify harder relationships as well.

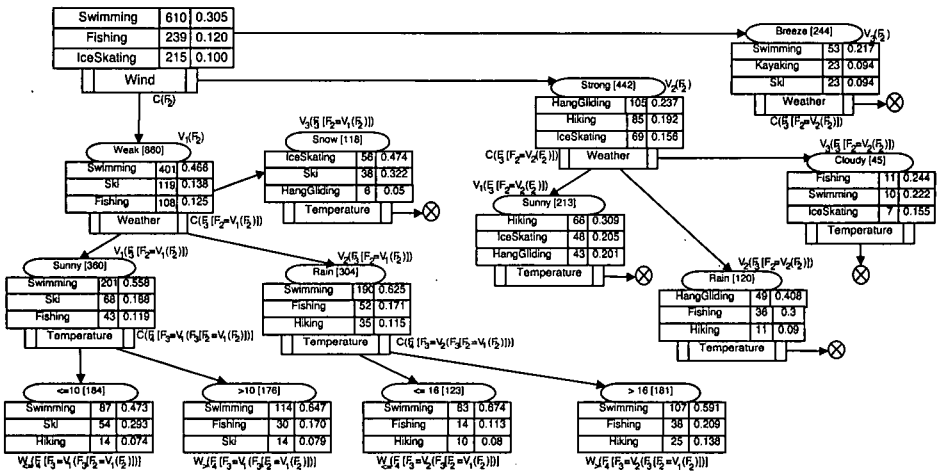


Figure 15: Sample tree in the Random Forest

4.1.5 jpPredefinedName

The *jpPredefinedName* plugin works on the assumption that many forms share field names and types. For example a field named *email* usually contains an email address which has to be in a valid email format. The plugin contains a list of constant names and their corresponding formats. This list is maintained and extended by the administrator of the Servlet.

4.1.6 jpRegExp

The regular expression plugin is geared towards learning regular expression values for fields. The plugin starts out by analyzing the historical values for the (F_1) field in particular its separator sign occurrence (e.g.: $-, +, @, (,), [,]$). This is built up from the assumption that form fields using regular expressions are usually finite and pre-defined in format. This means that a field will usually follow the same pattern historically if it belongs to the same form domain (e.g.: ISBN number, phone number, Social Security Number...etc). A statistical table is built up of these to determine any potential separator position recurrence. This helps identify possible separators for the field value's regular expression. It also lowers the processing time

of the algorithm as now only sets of fixed character lengths need to be checked. The plugin tries to match a separate regular expression for each section. We create a statistical tree which analyzes each section one character at a time. If there are no separators the algorithm will treat the complete field values as a single section. This will however cause uneven length inputs to offset the regular expression result (e.g.: if most inputs were 5 characters long and some were longer then the output can be something like $[A-Za-z]\{5\}[1-9ace]*$). If the range could not be merged into an optimal one then it will contain the subranges per character location (e.g.: $[a-c][f-k][A-Z]\{3\}$). In both section separated and single-section modes each step will try to optimize the ranges into smaller expressions to conserve space. The statistical table contains ratios and statistics on all positions and it will split only when the ratio for the separator is 100%. The separator identification has two modes: *fixed position* and *floating*. In case of the *fixed position* mode the segments are fixed in length as well as the position of the separators. The *floating position* mode has a dynamic position nature (e.g.: the @ sign in emails) in which case the only certain information the plugin has is the number of sections in all inputs.

If the separators and sections are identified correctly then each section is analyzed one position at a time using the similar approach to the above. Depending on the mode (fixed vs floating) the sections lengths are either constant length or dynamic. This however will only affect the expression normalization. For each position the possible values are collected and converted into regular expression ranges. After the end of each section the ranges in the actual section are compacted into a potentially shorter representation. This compaction includes replacing a range of $[0-9]$ to $[\backslash d]$ and ranges like $[abcghi]$ to a range of $[a-cg-i]$. Multiple occurrence of similar ranges or types are also checked and introduced (e.g.: $[abc][abc][abc]$ is converted to $[a-c]\{3\}$). Using a sample input of $(ab0\text{-}8cz, bc1\text{-}akm, dtt\text{-}d5e, cog\text{-}102)$ will generate an output of $[a-d][bcopt][01gt][-][18ad][05ck][2emz]$. In case of the *floating position* mode of the plugin we also utilize the $+$ and $*$ occurrence characters.

Once all segments have been "learned" the results are merged into one complete regular expression and matched against the remainder 50 percent of training data and if the ratio of the match is higher than the provided threshold then a rule is proposed. We have also experimented with reversing the logic of regular expression creation by starting out from the broadest ranges and tightening based on the results. This was also a good approach, however it provided more false positives due to the generic nature. The system also has an experimental regular expression plugin based on block-wise grouping and alignment algorithm coupled with a simple looping automata based on the concepts outlined in [19]. This algorithm is simplified by the additional information acquired from the potential separators acquired in the first pre-check step. We thought it was worth mentioning it in this section as it can provide a more optimal solution than the statistical approach.

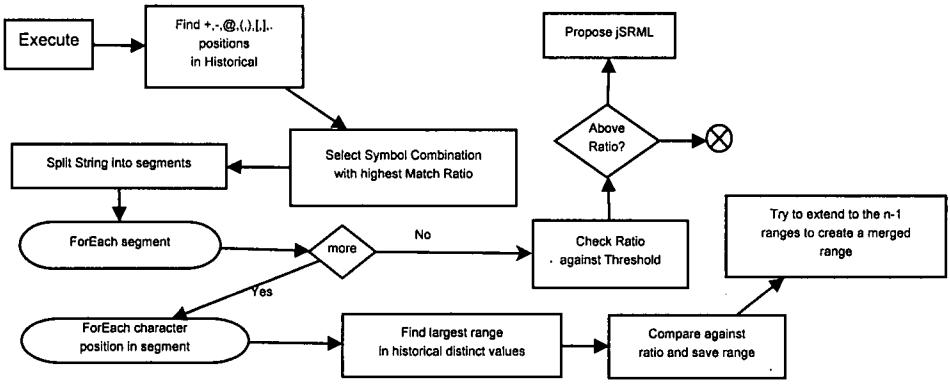


Figure 16: jpRegExp Plugin

4.2 Programatically evaluating the jSRML learning plugins

The jSRMLTool learning process uses a gradual approach to create the rules. The more positive inputs it receives the more effective the rules become. In order to provide a proper baseline it is advisable to feed in some positive form results. The results are summarized in *Figure 17* where T denotes True classification (including positive and negative), $F+$ means False positive and F- marks False negative with ES and PS marking Empty and Primed initial learning sets. The table includes the percentage results of the input classification (valid/invalid) for a specified plugin type. The learning is far from perfect, but with proper training it can aid the creation of validation rules. The simpler plugins like *jpFormat*, *jpLength*, *jpRange* are rather effective since they dynamically adjust their limits according to the inputs. The more complex plugins like the *jpRegExp* provided solid results, however it is more resource intensive and would take longer to provide the same success ratio. The *jpRelationship* plugin was excluded from the testing scenario as the random nature of the tests would not provide conclusive results on the efficiency of this plugin. We will demonstrate the real-life use of this plugin in a later section of this article.

During our tests we experimented with both empty and primed initial learning sets. In case of the empty learning set the number of false positives were considerably higher for the more complex plugins since they leveraged the distinct values and the learning set extensively. We did not run an evaluation on the *jpPredefinedName* plugin since that operates on a set of constant field names (e.g.: email, ip_address, isbn). The *jpCopyContent* plugin was also ignored for this evaluation since the results are based on equality between two fields and the random nature of the experiment offsets the actual findings of the plugin.

To test our plugins we used the following input sources:

- An English dictionary file containing *170,000* words. This is the source of all word subsets.

- A random list of *100,000* words from the dictionary to be used by the *jpLength* plugin.
- An email address list of *130,000* items built up from the dictionary with an added logic to generate valid/invalid emails. The ratio of valid/invalid emails was set randomly. The invalid emails were generated by adding known mistakes to words and symbols. The list also marks which are valid/invalid so that this information can be used in the validation evaluation. This is one of the sources of *jpRegExp*.
- A list of *50,000* phone numbers (matching US phone numbers: (CCC) NNN-MMMM) as the secondary input of *jpRegExp*.
- A list of *50,000* ISBN10 and ISBN13 random items as the tertiary input source for *jpRegExp*.
- A list of *50,000* IPV4 and IPV6 random items as an additional input for *jpRegExp*.
- A list of *250,000* regular expressions based on random expressions (variable in both format and length using $+, -, @, (,), [,]$. This will provide the additional learning set for *jpRegExp*.
- A list of *100,000* items randomly alternating between, *string*, *integer*, *double* and *date* for use with the *jpFormat* plugin.
- A list of *100,000* numbers between 1 and 1 billion. This list is used by the *jpRange* plugin.

Using the above sources we created *1,000* separate forms with random fields. Every form contained multiple fields (one to test each plugin). The *jpPredefinedName* and *jpCopyContent* plugins were ignored for the experiment. The reason why we chose to run the results on multiple forms was to ensure that the form fields and their contents were more random. For every field of the forms the test randomly selected the "expected" results of the validation. This was used to identify how successful the learning was. Each form was processed with *30,000* inputs with both *Empty* and *Primed* Set approaches to allow a better picture of the plugin efficiencies. The main operation flow of each set is as follows:

- `Empty Learning Set` : For each form randomly select *15,000* values from the corresponding lists for each field and run the engine on them. It must be noted that for this mode the engine cannot determine what the "expected" values are since the inputs are not classified. The engine will try to generate rules for what the "expected" values are by choosing an initial *15,000* inputs. These inputs are analyzed and a set of proposed validation rules are created based on the best fit using the ratios. Following this another *15,000* values are selected from the learning set and are used to observe the validation results.

Plugin	T ES	F+ ES	F- ES	T PS	F+ PS	F- PS
jpFormat	64.36 %	25.11 %	10.53 %	94.58 %	3.23 %	2.19 %
jpLength	59.65 %	22.18 %	18.17 %	88.09 %	7.17 %	4.74 %
jpRange	26.78 %	44.06 %	29.16 %	66.31 %	25.41 %	8.28 %
jpRegExp	29.59 %	36.17 %	34.24 %	51.57 %	21.12 %	27.31 %

Figure 17: Plugin comparison (ES=Empty Set, PS=Primed Set)

This is not an ideal approach since we cannot ensure that the first batch of inputs were completely valid therefore it will yield more false positives.

In case of the *jpRegExp* plugin the learning is not perfect due to the randomness of the selection. The remainder *15,000* values are run with each plugin and their classification is verified based on the expected versus the learned rules.

- `Primed Learning Set` : Using this approach the engine randomly selects *15,000* valid inputs for each field of each form based on the expected validation rules. As mentioned earlier every field has an "expected" validation requirement that is created during the form setup. The inputs might not fully overlap the expected target, however will be considered valid based on its definition. An example for *jpRange* would be an expected range of *[100,000-200,000]*. The random values that fit into the range will be considered valid and will allow the plugin to create its own jSRML rule suggestion. Due to the random selection of valid elements a learned range for the previous criteria might be *[125,000-170,000]* (which is a subset of the original "expected" range). In case of the *jpFormat* plugin items with the expected format (string, integer, date, double) are selected from the list as the initial set. This will be the "valid" set of inputs. In case of *jpRegExp* one of eight predefined expression formats are selected as the "expected" validation rule and values that match this format (these formats are: email, ipv4, ipv6, phone, isbn10, isbn13, webaddress, phone). Afterwards a remainder *15,000* inputs are selected and executed using the rules. During the processing of the remainder inputs the engine checks the learned rule results with the expected classification. Using these we are able to measure the efficiency of the learning.

The results of the forms are averaged and evaluated in *Figure 17*. Based on the results it is visible that using *Primed* Sets yields the most effective results. From the plugins *jpFormat*, *jpLength* and *jpRange* yield the best results. The regular expression matching *jpRegExp* plugin does provide good results, however the evolution of the format recognition should be tuned in the future. It should be noted that the current efficiency of the implemented plugins are not at 100%. This can lead to a valid question: how do we validate a form that is only *n%* effective? The short answer is that the acceptance threshold should be set so that the domain owner can accept the efficiency of the results. Even if the results are not 100% it still

provides a direction to better tune the validation requirements. The more examples the engine can derive decisions and learn from the higher the efficiency becomes. In a human oriented approach the fields have more relationship and are chosen based on some expected behavior. One might argue if the whole learning validation rules has any relevance in the forms nowadays. We believe that the jSRML language provides a cleaner and more powerful way to define form validation rules. Allowing the option to intercept and potentially learn validation rules in a non-obtrusive way not only allows administrators with a powerful tool to create rule but can also be used to mine the inputs based on the submissions and potentially discover relationships and visitor decision patterns in the submitted form.

Due to the random nature of the previous experiment we felt it would be worthwhile to demonstrate an incremental approach as well for some of the plugins to better observe how the ratios change by gradually introducing more and more positive examples to the experiment. We chose a significantly smaller, more targeted learning set to better demonstrate how the plugins learn the results. This more constrained testbed yielded considerably better results.

For the *jpRegExp* we used a regular expression of $[1-4A-Za-z]\{5\}[-][1-6]\{5\}[-][a-k][A-P]\{8\}[-][1-9A-Za-z]\{8\}$ as the valid format (example valid inputs are: *1QrHk-56566-bPFI1ENNL-TLKir5Qk* and *h2bwM-61632-fELCGFJEM-631237Va*). This is a simpler regular expression then an *email* or *conditional isbn number* expression (matching both *isbn10* and *isbn13* formats), but still provides adequate ground to demonstrate how the plugin's efficiency evolves in proportion to the number of positive examples. The input examples for *jpRegExp* are 30 characters long and randomized on each character so we don't really need a set of tens of thousands of positive examples to learn them.

During the experiment the *jpRange* target range was also reduced to a smaller magnitude. The experiment sets a random range between *100* and *5,000*. The *jpLength* target was randomly selected with an upper limit of *400*, causing the experiment to terminate around *400-500* positive examples with a 100% ratio.

We provide *100* valid inputs for each plugin at the start of the test. We then take *50* positive and *50* negative for each plugin and observe how the rules classify the results and record the incorrect/missed classification counts. We are able to ensure that the the training examples are positive and negative since we select them according to our predefined criteria. We perform this over ten iterations. In each iteration we increase the positive examples by *100* and regenerate the validation rules. These rules are then run against *50* more positive and *50* more negative examples. After the tenth iteration we are priming the experiment with *1,000* positive examples and testing against *500* positive and *500* negative examples. This is a very controlled experiment but it is useful to demonstrate how the ratios converge in proportion to the number of training examples. The results of the experiment can be seen in *Figure 18*. It can be seen in the figure that with proper and controlled positive inputs the plugins can provide near 100% ratios as well. In the next section we will demonstrate a real-life example where these results can be put into practice.

Analyzed Plugin	Total Examples	Miss Count	Success Ratio
jpLength	100	17	83.00 %
jpLength	200	7	96.50 %
jpLength	300	2	99.33 %
jpLength	400	0	100.00 %
jpRange	100	72	28.00 %
jpRange	200	85	57.50 %
jpRange	300	97	67.67 %
jpRange	400	81	79.75 %
jpRange	500	63	87.40 %
jpRange	600	59	90.17 %
jpRange	700	45	93.57 %
jpRange	800	34	95.75 %
jpRange	900	28	96.88 %
jpRange	1,000	11	98.90 %
jpRegExp	100	98	2.00 %
jpRegExp	200	186	7.00 %
jpRegExp	300	198	34.00 %
jpRegExp	400	146	63.50 %
jpRegExp	500	90	82.00 %
jpRegExp	600	48	92.00 %
jpRegExp	700	22	96.85 %
jpRegExp	800	12	98.50 %
jpRegExp	900	6	99.33 %
jpRegExp	1,000	2	99.80 %

Figure 18: Plugin Efficiency with gradual positive training examples

4.3 A Real-world example: Dentistry Treatment Enquiry Form

To evaluate the engine further we have hooked up the jSRMLTool servlet to an already functioning form to verify what the engine suggested for the validation rules. This was a more exhaustive test than the previous outdoor activity survey. In the earlier example we only demonstrated the engine use for the *jpRelationship* plugin. During this test more plugins of the engine were verified as a whole. We chose [20] which is a site targeted at capturing leads for international clients who are enquiring about dental treatment in Hungary. The booking form contained several fields providing an ideal fit to test some of the plugins. Using the site's form we were tested: *jpFormat* (*Age*, *Country* field), *jpRange* (*Age* field), *jpRegExp* (*Phone* field), *jpPredefinedName* (*Email* field), *jpLength* (*First name*, *Last name*, *Phone*, *Treatment*, *How may we help* fields), *jpCopyContent*(*Confirm* email). The *Treatment* field was used in conjunction with the *Age*, *Gender* and *Country* fields to perform a *jpRelationship* conditional learning. The *Treatment* field had multiple non-conflicting rules generated using different plugins. The system found the range of the length used for the input and also used it for the conditional learning.

Our experiment used the site's historical data for lead submissions and ran *537* leads acquired form the site using *Selenium*[21] (scriptable automated tester framework) to emulate the form posting. The results were impressive, since it was able to provide effective validation rules for most fields. The *Phone* field had some weak rule recommendations (e.g.: $[0-4][1-5][0-9]+$), however the ratios were not

Field	Validation Results	Plugin
Age	[35, 70]	`jpRange`
Age	integer	`jpFormat`
Country	$5 < length < 12$	`jpLength`
First Name	$4 < length < 7$	`jpLength`
Last Name	$4 < length < 10$	`jpLength`
Email	email	`jpPredefinedName`
Confirm Email	Email match	`jpCopyContent`
Gender	$4 < length < 6$	`jpLength`
Phone	string	`jpFormat`
Phone	$7 < length < 14$	`jpLength`
Treatment	$4 < length < 37$	`jpLength`
Treatment	conditional	`jpRelationship`
How may we help?	$5 < length < 184$	`jpLength`

Figure 19: Plugin results for Dentistry Contact form

high enough due to entries with hyphens and extension numbers along with entries starting with + for international exit codes. Since the target ratio was not breached the plugin's rule recommendation was ignored. The output of the validation rules for each plugin can be seen in *Figure 19.*

The experiment yielded in providing validation rules based on the results visible in *Figure 19.* Most of the plugins yielded considerable adaptive results. If we would run the forms with more training examples then the ranges and results would improve as well. The experiment also showed that *"55% of clients requesting All-on-four dental as their treatment are Male, over the age of 50 and live in the UK."* (which is identical to the statement: In 55% of the cases "Treatment" is "All-on-four-dental" when the client is "Male" is from the "UK" has an age above "50"). The learning results also showed that *"61% of Abutment related requests come from Female clients from Ireland who are under 60".* This provided a good demographic analysis of the visitors and helped the site adjust their marketing strategies accordingly. Even though data mining was not the focus of the experiment it did provide a direction for future study for the jSRML engine. The experiment proved the viability of such a solution in a real world scenario. The learning is not yet perfect, the rule engine and concept of allowing easier rule definitions substantially outweigh the performance and efficiency shortcomings (which can be tuned by introducing better learning plugins into the system).

5 Related Work

There have been several advances and research done in the field of form validation. In this section we will mention a few along with how the approaches handle validation. The first paper we would like to mention is [22]. This article proposes the use of an XML based rule definition to show field validation. They create an XML file based on the database model itself on both the *Client-* and *Server-side* level. While it is a good approach it still lacks the flexibility of the user overriding and defining custom conditions. In many cases structural and type validity is not

enough, context validity should also be considered. This means that even though a field's value is correct it might have dependencies on other fields which are not visible on a database schema level. The approach lacks the option to provide custom hooks and does not provide provisions for data correction.

Another paper that we would like to mention is [23]. This article proposes that the validation of forms should be part of the model design and handled on the server-side. They leverage Spring MVC as part of their AC-MDSD (Architecture Centric Model Driven Software Development). Although a good approach it requires the form validation to be coded as part if the datamodel on the server that will process the data. Our jSRMLTool's server mode provides a more comprehensive set of features and does not force the developer to predefine their dataset prior to deploying the processing application.

The next approach we would like to mention is [24]. This proposes a rule based field validation using JavaScript. The rules themselves are basic but support the comparison and aggregation of multiple field values. The validator engine itself does not have any hooks and does not allow the user to control what should happen if the validation fails. Our approach offers a solution to both and provides a way to dynamically correct the field data making it a very powerful tool.

The authors of [25] propose an automatic Server-side validation approach for HTML forms. It collects the form elements and stores the validation elements inside a database and provides an interface for the administrators to go in and specify how to validate the given fields. Currently they do not offer too complex validation methods (since the approach is mainly focused on type and format oriented validation). It does not offer dependency or regular expression definitions for the field values. It does bare some similarities to what we wish to achieve with the Servlet mode of our engine. Our library not only offers the forms to be validated using a centralized server, but also provide the definition of more complex validation rules.

The points discussed in [25] are aimed at server-side validation and are valid for most web forms. The article does however suggest that people will disable JavaScript which would render client-side validation useless. This is a long and heated debate in the web community as most modern web pages utilize JavaScript and flash excessively. Disabling JavaScript support will not only render the validation useless but also hinder the usability of the page itself.

We should also mention the approach presented in [26]. This paper introduces a language called EEL (edit engine language) to provide a common way of describing field validation rules. This language was applied in the telecommunications area where several forms were being submitted. Although their approach was aimed at non-HTML forms, and was written purely in C++ it does have a solid syntax and could potentially be extended to be used in a modern web solution (after porting it to JavaScript or a server-side language).

The ideas raised in [27] demonstrate a .NET approach to rule based form validation. It also uses an in-line approach similar to jSRML. The rules can have conditions and it supports regular expressions as well. The rules are not as readable as jSRML and do not provide support for context related rules. For the rule definition it allows the reference of only one other field rather than providing a

complete context based approach. It does provide a solid solution for .NET based forms which we believe is worth investigating in the future. Our metalanguage is not limited to one technology stack or implementing language so creating a .NET library isn't hard to envision and implement.

6 Summary and future work

In this article we introduced the jSRML metalanguage and engine. This is a major extension to the SRML language specification to allow it to be used in the form validation space. After showing the background technologies and demonstrating how form validation works we provided the jSRMLTool engine. Our engine allows both *Client-side* and *Server-side* validation modes using the jSRML language. The extension allows non-obtrusive definition of form validation rules. The jSRMLTool engine can also correct the form values making it extremely useful in situations when the submission can contain errors that can be corrected based on rules. We also showed ways to provide real-time validation. Our tool also helps in the generation of jSRML rules using machine learning. The rules can change over time based on the form inputs. We believe jSRML is a valuable asset in the ever-growing pursuit for providing pristine and valid data acquired from web forms.

In the future we plan to simplify the rule definitions simplifying the syntax and by providing more out of the box types (to avoid longer binary-op conditions). We also plan to extend the servlet service to allow easier updating and creation of rules for server-side validation alongside enhancing the learning module to provide more powerful and efficient rules. We plan to investigate the option to generate test data based on the rules defined in the form file to help test driven development as well as exploring other languages aside from Java for the library implementation.

References

[1] Raggett, D., Hors, A. L. and Jacobs, I., 1998, *"HTML 4.0 specification," W3C*, http://www.w3.org/TR/REC-html40/

[2] Handley, M., 2006, *Internet Denial-of-Service Considerations*, IAB, RFC4732

[3] Boyd, S.W. and Keromytis, A.D., 2004, *SQLrand: Preventing SQL injection attacks.* International Conference on Applied Cryptography and Network Security (ACNS), LNCS, volume 2, 2004.

[4] Crockford, D., 2008, *Javascript: The Good Parts.* O'Reilly, 2008.

[5] Garret, J.J, 2005, *Ajax: A New Approach to Web Applications*, http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[6] Lindley, C., 2009, *jQuery Cookbook.*, O'Reilly Media

[7] Havasi, F., 2002, *XML Semantics Extension*, Acta Cybernetica Vol 15 No. 2, pages 509-528

[8] Hégaret, P., 2005, Document Object Model (DOM), W3C, http://www.w3.org/DOM/

[9] Kálmán, M., and Havasi, F. et al, 2006, *Compacting XML documents.*, Journal of Information and Software Technology, Volume 48 Issue 2, February 2006, pages 90-106

[10] Bray, T., Paoli, J. and Sperberg-McQueen, C., 1998, *Extensible markup language*, XML 1.0 W3C recommendation, http://www.w3.org/TR/REC-xml

[11] Kálmán, M., 2013, *The complete XSD of jSRML* http://www.srml-language.com/jSRML/jSRML.xsd

[12] Hunter, J. and Crawford, W., 2001, *Java Servlet Programming.* O'Reilly, 2nd edition edition, 2001.

[13] Clark, J. and DeRose, S., 1999, *XML Path Language (XPath) Version 1.0.* http://www.w3.org/TR/xpath

[14] Lie, H.W. and Bos, B., 1999, *Cascading Style Sheets, designing for the Web*, Addison Wesley

[15] Kálmán, M., 2013., *The complete HTML source of the example* http://www.srml-language.com/jSRML/jSRML-example.txt

[16] Hastie, T., Tibshirani, R. and Friedman, J., 2001, *The Elements of Statistical Learning*, Springer. ISBN 0–387–95284–5.

[17] Breiman, L., 2001., *Random forests.*, *Machine Learning*, 45:5–32, 2001.

[18] *SurveyMonkey Online Surveys*, 2013, http://www.surveymonkey.com

[19] Fernau, H., 2009., *Algorithms for learning regular expressions from positive data*,Inf. Comput., Volume 207, Number 4, pages 521–541, Academic Press, Inc.,Duluth, MN, USA

[20] Beauty and Confidence, 2013, *Dental Implant Abroad Booking page* http://www.dental-implantabroad.co.uk/ental-implant-overseas/

[21] SeleniumHQ, 2013, http://docs.seleniumhq.org/

[22] Liang, Z., 2009, *A field-oriented approach to web form validation for Database-Isolated Rule*, Man and Cybernetics, SMC 2009. IEEE International Conference on Systems, 11-14 Oct. 2009, pages 4607–4612

[23] Escott, E., Strooper, P., et al, *Model-Driven Web Form Validation with UML and OCL*, 2012, Lecture Notes in Computer Science Volume 7059, 2012, pages 223-235

[24] HansMartin, A., 2010, *Form validation with Rule Bases* http://blog.mgm-tp.com/2010/10/test-data-generation-part1

[25] Saha, T.K. and Ambia, A., 2013, *Code Generation Tools for Automated Server-side HTML form Validation*, International Journal of Computer Science and Management Research, Volume 2, Issue 1, 2013, pages 1265–1271

[26] Blando, L., 1999., *A Framework for a Rule-Based Form Validation Engine* http://wiki.lassy.uni.lu/Special:LassyBibDownload?id=324

[27] Giannoudis, J., 2012., *Rule Based Validation for ASP.NET* http://www.codeproject.com/Articles/367214/Rule-Based-Validation-for-ASP-NET

Received 23rd May 2013

Acta Cybernetica **21** (2014) 585–607.

Connection Between Version Control Operations and Quality Change of the Source Code

Csaba Faragó*, Péter Hegedűs†, Ádám Zoltán Végh*, and Rudolf Ferenc*

Abstract

Software erosion is a well-known phenomena, meaning that software quality is continuously decreasing due to the ever-ongoing modifications in the source code. In this research work we investigated this phenomena by studying the impact of version control commit operations (add, update, delete) on the quality of the code.

We calculated the ISO/IEC 9126 quality attributes for thousands of revisions of an industrial and three open-source software systems with the help of the Columbus Quality Model. We also collected the cardinality of each version control operation type for every investigated revision. We performed Chi-squared tests on contingency tables with rows of quality change and columns of version control operation commit types. We compared the results with random data as well.

We identified that the relationship between the version control operations and quality change is quite strong. Great maintainability improvements are mostly caused by commits containing Add operation. Commits containing file updates only tend to have a negative impact on the quality. Deletions have a weak connection with quality, and we could not formulate a general statement.

Keywords: Software Maintainability, Software Erosion, Source Code Version Control, ISO/IEC 9126, Case Study

1 Introduction

Software quality plays a crucial role in modern development projects. There is an ever-increasing amount of software systems in maintenance phase, and it is a well-known fact that software systems are eroding [15], meaning that in general their quality is continuously decreasing due to the ever-ongoing modifications in its

*University of Szeged Department of Software Engineering, Árpád tér 2. H-6720 Szeged, Hungary, E-mail: {farago,azvegh,ferenc}@inf.u-szeged.hu

†MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary, E-mail: hpeter@inf.u-szeged.hu

source code, unless explicit efforts are spent on improvements [3]. Figure 1 shows this phenomena.[1]

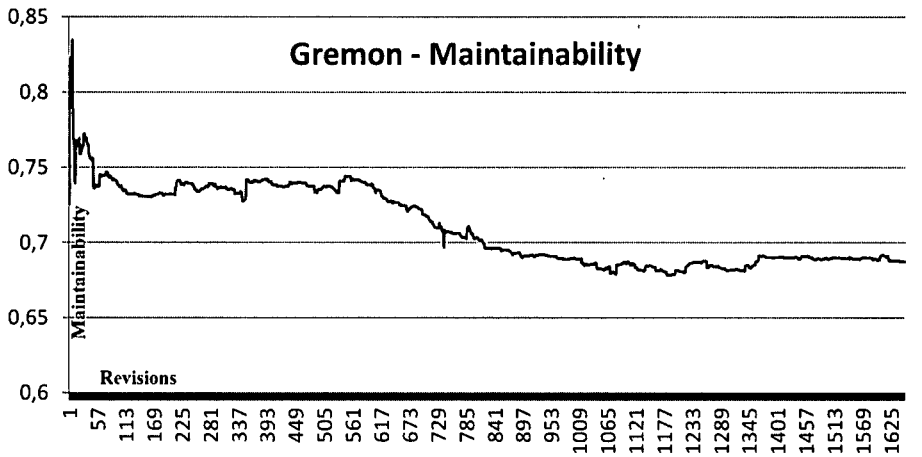


Figure 1: Maintainability values of the Gremon project

Our aim is to check the connection between the developers' interactions and the quality change. These interactions can be the following: version control operations, development-time IDE interactions and interactions in the issue tracking system. We are motivated to perform this research for several reasons. Determining typical patters which have significant effect on maintainability could help us better allocate software developer efforts. For example, a more strict code review is necessary for those commits which have statistically higher impact on maintainability. On the other hand, we expect that in longer term we will be able to find typical patters, which are bad habits of developers, and eliminating these could have a positive impact on maintainability. We especially expect such findings from the analysis of IDE interactions.

For this first step, we checked the version control operations only; and within this set of information we focused exclusively on the mere number of various operations, i.e. how many files were added, updated and deleted within that commit. Other commit-related information, like the certain files affected, the change itself, the comment, the date or the author, are not considered in this research.

Figure 2 illustrates how this step fits into our longer-term research goals. The general research field is to study how the developer interactions (illustrated with trapezoids) affect various software characteristics (within the rectangle). We identified 3 data sources from which data about developer interactions can be extracted: (1) the version control operations, (2) the IDE micro interactions, and (3) the data found in issue tracking systems. The version control operations are preceded by IDE micro interactions, i.e. the developer typically performs several IDE actions

[1]Gremon is one of the four software systems used in this research. See 4.1 for more details about this project.

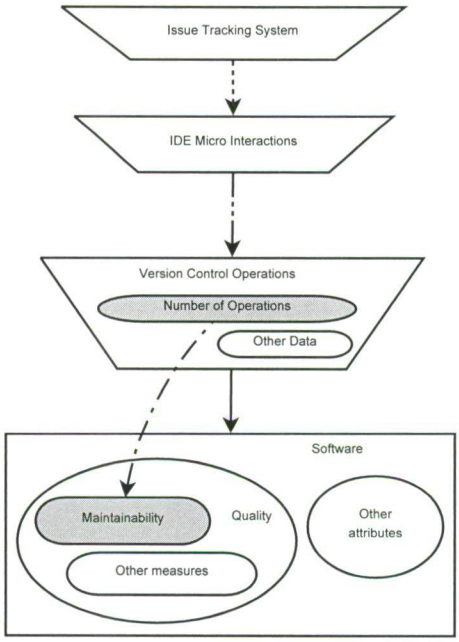


Figure 2: Overview

before committing changes. The most exiting area of research would be the IDE interactions; however, hardly any available data exist, and these are incomplete in most cases. Furthermore, it is not trivial to assign these interactions to concrete commits, therefore we decided that in the first period we concentrate on version control interactions only. Data found in the issue tracking system could be also interesting in longer term, provided that they contain substantive data about the reality. From the software attributes we selected software quality as worth for investigation at first, and among its subcharacteristics, we decided to study maintainability. The figure provides an overview about the possible directions of future investigations.

This and possibly other future results could help identifying the typical patterns where code erosion occurs. These patterns could be very useful information for proactive action planning: to find a better distribution of the efforts intended for code quality improvements.

We were motivated by the question: does the way of introducing code changes (reflected by version control operations of different commits) have a traceable impact on software quality? Do all types of commit operations contribute to software erosion, or are there exceptions?

For the definition of software quality we refer to the ISO/IEC 9126 standard [8], which defines six high-level characteristics that determine the product quality of software: *functionality, reliability, usability, efficiency, portability,* and *maintain-*

ability. Due to its direct impact on development costs [3], and being in close relation with the source code, maintainability is one of the most important quality characteristics.

The types of the version control operations and the maintainability of the code are at first glance remote concepts, more or less independent from each other. Furthermore, as no finer grained information is considered at this point (e.g. what was changed in the file, who made the change, or even on which file the change was performed), the distance between the maintainability change and the commit operations is even higher. Therefore, it is a non-trivial question if there is any connection between the two datasets at all.

Supposed that there is a connection between version control operations and maintainability changes in case of each examined projects, we are interested in finding out which are the common patterns, i.e. those connections which are significant for every examined project. These can be formed as general statements.

By performing experiments we tried to find evidences which support or reject some of our more concrete assumptions based on Figure 1. The beginning of the time line is very hectic. This is the start of the project with many additions of new parts. The maintainability becomes smoother later on, and the long-term tendency is negative. This is the phase when modifications on the existing sources are performed, and less new sources are added. Furthermore, based on our experiences, developers tend to pay bigger attention on the quality when adding new code than updating it later due to e.g. bug fixing, and this is especially true for the code originally developed by someone else. It is a hard task in itself to understand the code, reproduce the error, debug and find the solution, therefore developers under time pressure are glad if they find a solution; finding a nice solution is often not reached.

Based on the above explained expectations we formulated the following research questions:

- **RQ1:** *Do commits containing file additions to the system have a significant positive impact on its maintainability?* Our assumption is that they have, as they introduce new, clean, reasoned code.
- **RQ2:** *Is it true that the commits containing source file updates only tend to significantly decrease the maintainability?* Our assumption is that it is, as the ongoing development activity – without planned improvements in quality – tends to decrease maintainability, and having only file updates is a sign of this phase of the software development.
- **RQ3:** *Do commits containing file deletion improve the maintainability of the system?* Our assumption is that they do, as file deletions could be a sign of refactoring; therefore, better maintainability is expected if there is such an operation present in the commit.

The paper is organized as follows. Section 2 introduces works that are related to ours. Then, in Section 3 we present the methodology used to test the underlying relationship between version control operations and maintainability changes.

Section 4 discusses the results of the performed statistical tests and summarizes our findings. In Section 5 we list the possible threats to the validity of the results, while Section 6 concludes the paper.

2 Related Work

The version control system and other types of development related repositories (e.g. bug tracking system) provide a rich source for data mining approaches. These approaches can be used for collecting different kinds of process metrics, identify bug introducing or bug fixing changes, create bug prediction models, etc. In the presented paper we focus on finding traceable evidences of the relationship between the changes in software maintainability and the different types of version control operations in developer commits; but first, we collect the works dealing with similar researches to ours.

There are works which focus on the effect of software processes to the product quality [10]. Hindle et al. [7] deal with understanding the rationale behind large commits. They contrast large commits against small commits and show that large commits are more perfective, while small commits are more corrective. Bachmann and Bernstein [4] explore among others if the process quality, as measured by the process data, has an influence on the product quality. They showed that the product quality – measured by number of bugs reported – is affected by process data quality measures.

There are also others who utilize process metrics to detect failure-prone components of the software [9, 12]. Nagappan et al. show that applying different process metrics significantly improves the accuracy of the fault-prone class prediction [14]. They also present an empirical case study [13] of two large-scale commercial operating systems, Windows XP and Windows Server 2003, where they leverage various historical in-process and product metrics to create statistical predictors to estimate the post-release failures. We think that the number of defects revealed in the code is only one aspect of maintainability. Moreover, our aim is not to predict fault-prone parts of the source code, but to get a general picture about the effect of the way changes are introduced (i.e. version control operations in the commit) to software maintainability.

Lots of works build models for predicting refactorings based on version control history analysis [18, 19]. Moser et al. [11] developed an algorithm for distinguishing commits resulted by refactorings from those of other types of changes. Peters and Zaidman [16] investigate the lifespan of code smells and the refactoring behavior of developers by mining the software repository of seven open-source systems. The results of their study indicate that engineers are aware of code smells, but are not very concerned by their impact, given the low refactoring activity.

There are also papers that try to reveal the change-proneness of different source code elements [20, 23] based on version control history. Giger et al. [5] explore prediction models for whether a source file will be affected by a certain type of source code change. For that, they use change data of the Eclipse platform and the

Azureus 3 project. Ying et al. [21] have developed an approach that applies data mining techniques to determine change patterns – sets of files that were changed together frequently in the past – from the change history of the code base. Our focus is not on introducing a new sophisticated repository mining technique and applying it for some kind of prediction. We use the number and types of different version control operations and examine the effect they have on software maintainability.

In this research we analyzed Java source code, as the used quality model handles that programming language. A quality model for C# was presented by Hegedűs [6].

3 Methodology

This section summarizes the types of collected data during the experiment and describes the methodology of analyzing them. Particularly, we describe what we exactly mean under version control operations and maintainability change, and the methodology used to analyze the data.

3.1 Version Control Operations

In this work we investigated the number of various version control operations of the examined commits. Only the mere numbers of various operations were considered, e.g. 2 files were added, 5 files were updated and 1 file was deleted within the examined commit. We omitted every other version control-related data, e.g. the date, the names of the affected files, the author of the file, or the comment of the files. These data will be used for finer-grained analysis in the future.

We analyzed only Java source files, so we skipped all other types of file system entries like directories or non-Java files (e.g. xml files). We did this because the current version of the used quality model considers only the Java source files. Besides *Add*, *Update*, and *Delete*, there is a fourth version control operation: *Rename*. As there were hardly any Rename operations in the examined data (it occurred only in one of the analyzed projects with very low cardinality) this operation was not considered. Therefore, the input data collected from the version control system was an integer triple for each commit containing at least one Java source file:

- **A** - the total number of file additions,
- **U** - the total number of file updates,
- **D** - the total number of file deletions.

3.2 The Applied Quality Model

To calculate the absolute maintainability values for every revision of the systems we used ColumbusQM, our probabilistic software quality model [2] that is based on the quality characteristics defined by the ISO/IEC 9126 [8] standard. The computation of the high-level quality characteristics is based on a directed acyclic

graph (see Figure 3) whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). Internal quality properties characterize the software product from an internal (developer) view and are usually estimated by using source code metrics. External quality properties characterize the software product from an external (end user) view and are usually aggregated somehow by using internal and other external quality properties. The nodes representing internal quality properties are called *sensor nodes* as they measure internal quality directly (white nodes in Figure 3). The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. In addition to the aggregate nodes defined by the standard (dark gray nodes) we also introduced new ones (light gray nodes).

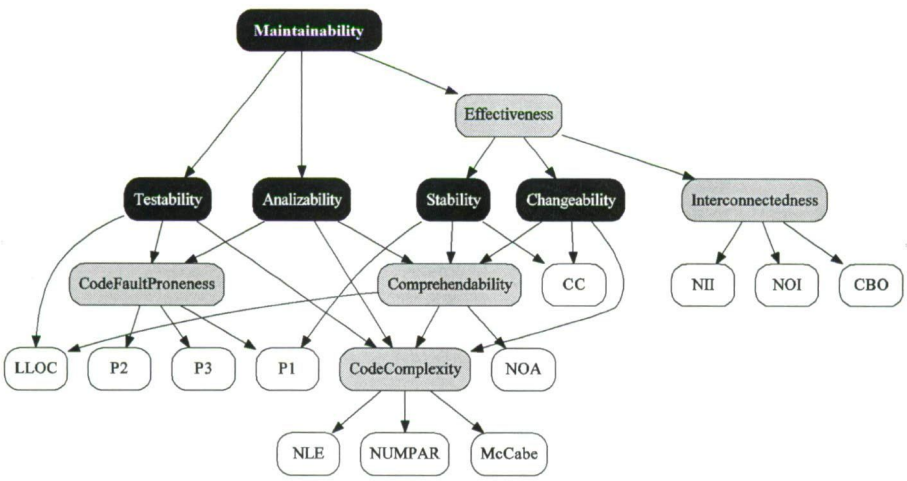


Figure 3: ColumbusQM – Java ADG

The current version of the model applies the following source code metrics:

- *LLOC (Logical Lines Of Code)* – the LLOC metric is the number of non-comment and non-empty lines of code.
- *NOA (Number Of Ancestors)* – NOA is the number of classes that a given class directly or indirectly inherits from.
- *NLE (Nesting Level Else-if)* – NLE for a method is the maximum of the control structure depth. Only *if, switch, for, foreach, while, and do...while* instructions are taken into account and in the if-else-if constructs only the first if instruction is considered.
- *CBO (Coupling Between Object classes)* – a class is coupled to another if the class uses any method or attribute of the other class or directly inherits from it. CBO is the number of coupled classes.

- *CC (Clone Coverage)* – clone coverage is a real value between 0 and 1 that expresses what amount of the item is covered by code duplication.
- *NUMPAR (NUMber of PARameters)* – the number of parameters of the methods.
- *McCC (McCabe's Cyclomatic Complexity)* – the value of the metric is calculated as the number of the following instructions plus 1: *if, for, foreach, while, do-while, case label* (which belongs to a switch instruction), *catch, conditional statement (?:).*
- *NII (Number of Incoming Invocations)* – the number of other methods and attribute initializations which directly call the method. If a method is invoked several times from the same method or attribute initialization, it is counted only once.
- *NOI (Number of Outgoing Invocations)* – the number of directly called methods. If a method is invoked several times, it is counted only once.
- *WarningP1/P2/P3 (Serious/medium/minor coding rule violations)* – the number of serious/medium/minor PMD (`http://pmd.sourceforge.net/`) rule violations in the class.

The edges of the graph represent dependencies between an internal and an external or two external properties. The aim is to evaluate all the external quality properties by performing an aggregation along the edges of the graph, called Attribute Dependency Graph (ADG). We calculate a so called *goodness value* (from the [0,1] interval) for each node in the ADG that expresses how good or bad (1 is the best) is the system regarding that quality attribute. The probabilistic statistical aggregation algorithm uses a so-called benchmark as the basis of the qualification, which is a source code metric repository database with 100 open source and industrial software systems. For further details about ColumbusQM, see our previous work by Bakota et al. [2].

3.3 Contingency Table

The contingency table is a two-dimensional table with the maintainability changes in the rows and version control operation categories in columns, and the cells containing the total number of commits in the category causing that kind of maintainability change.

The maintainability changes were partitioned into three sets:

- +: positive change,
- 0 : no traceable change,
- −: negative change.

The maintainability change is positive if the calculated value of the actual commit is higher than the value of the previous commit, negative if it is lower and 0 if the two values are the same.

The commits were divided into several disjoint categories based on the version control operations they include. The categories were defined based on intuition coming from the principal component analysis (PCA) of the industrial project's data set. We defined the following categories:

- D: commits containing at least one *Delete* operation,
- A: commits containing no *Delete* operation, containing at least one *Add* operation,
- $U+$: commits containing *Update* operations only; the number of *Update* operations is at least 2,
- $U1$: commits consisting of exactly one *Update* operation.

Please note that the union of these commits is the full set of examined commits. Commits affecting no Java files do not have any effect on the calculated maintainability, therefore they were omitted from the calculation.

3.4 Bar Plot Diagrams

In order to visualize the data found in the contingency tables we used proportional bar plot diagrams (see e.g. Figure 5). Each commit category is represented by a bar, which is divided into 3 parts: the proportion of positive, zero and negative maintainability changes within that category. For a better comparison the proportions of the full commit set are also presented.

We can also get intuitions about the answers of the research questions based on these diagrams. If there are spectacular differences among categories within a project, and there are similarities in the diagrams among projects, then it suggests that the connection between the version control operation types and the maintainability is quite strong, and it even adumbrates the answers on some of the research questions.

3.5 Contingency Chi-squared Test

To give well-grounded answers to our research questions we performed Chi-squared tests [1] (similarly to the method presented by Ying and Robillard [22]) on the contingency tables.

This test calculates the expected values based on the sum of rows and columns, i.e. what were the values if there were no connection between version control operations and maintainability. Then it determines if the differences between the actual and the expected values are significant or not. The null-hypothesis is that these values are the same, and the reason of the differences are random. The final

result of this test is practically the p-value, indicating the chance of the result being at least as extreme as the observed, provided that the null-hypothesis is true.

The test was performed using the `chisq.test()` R function [17]. This function calculates the standard residuals (*stdres*) as well for each cell, i.e. what would the value be if the data were of standard normal distribution. E.g. if this value was -2.0, then it would mean that the number of the observed elements was less than the expected (see the negative sign), and the difference was as much likely to be random as a standard normally distributed variable is at least as extreme as 2.0 (i.e. less than -2.0 or greater than 2.0).

Based on these standard residuals the p-value is calculated as follows. The R function `pnorm()` calculates the distribution of the given values, i.e. the proportion of elements less than or equal to the provided one. E.g., this value is 0.5 for 0.0, 0.023 for -2.0, 0.977 for 2.0 etc. Based on the definition of the p-value, the result for value 0.0 would be 1.0, i.e. there is no deviation from the expected value at all. To go on with the running example, for -2.0 we need to calculate the proportion left to -2.0 and right to 2.0, and sum it. As mentioned, the first value is 0.023, while the second one is also 1.0-0.977=0.023. Therefore the p-value is 0.046.

This process is illustrated on Figure 4. The size of both gray areas is 0.023. The lower dashed line is at 0.023, while the upper one is at 0.977.

To summarize, we have the following formula for calculation:

$$2 \cdot pnorm(-abs(x))$$

where x is the value of standard normal distribution. The cells containing small p-values can be considered as significant results.

In order to provide a quick and easy overview of the results, one last step was performed: the number of zeros between the decimal point and the first non-zero digit of the p-value were calculated, with the appropriate sign, denoting the direction of the deviation from the expected value (negative if it is less than the expected, positive if it is greater). More formally, if the canonical form of the p-value is $(a \cdot 10^b)$, the transformed value is the absolute value of the exponent minus one (i.e. $|b| - 1$), with the sign of the standard residual. E.g., in the above example the p-value in canonical form is $4.6 \cdot 10^{-2}$, and the sign of -2.0 is negative, therefore the transformed value is -1. 0 means that the random probability is at least 10%, 1 and -1 means that it is between 1% and 10% and so on. Formally, this transformation was calculated by the following function:

$$f = \left\lfloor log\frac{1}{p} \right\rfloor \cdot sign(stdres)$$

This test also gives a common p-value, i.e. not only cell based p-values. Having a low enough such p-value ($p < 0.01$) would answer positively the base question if there is a connection between version control operations and maintainability.

For answering the research questions formally, we take the last, transformed table. In case of the cell-based approach we consider those values significant, where the absolute values are at least 2 ($p < 0.01$) for all of the checked software systems.

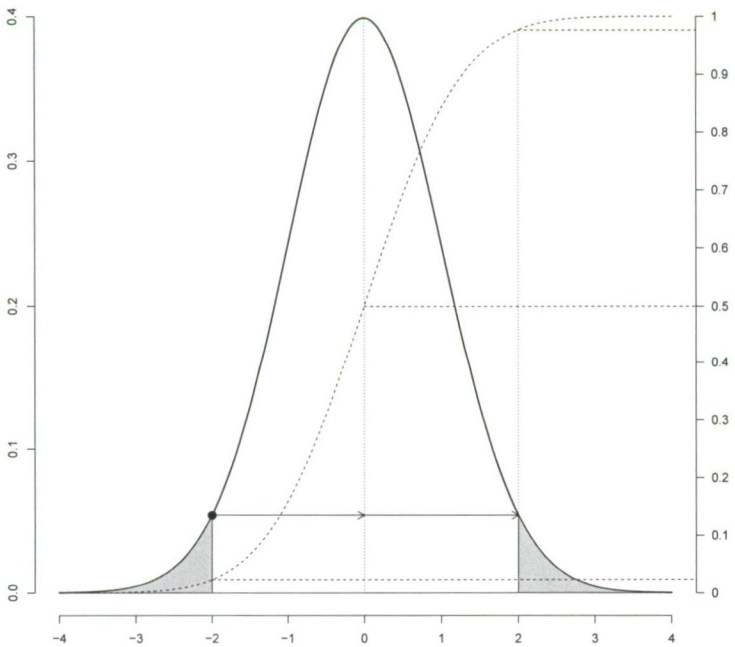


Figure 4: Standard normal distribution

3.6 Random Checks

To validate the results, a random analysis was performed as well. This was done in the following way:

- We kept the source control operation data as it was.
- We also kept the values of the quality changes, but we permuted randomly the order of the revisions it was originally assigned to, just like a pack of cards. The `sample()` R function was used to permute the order.

We performed randomization several times, permuting the already permuted series. We executed the same analysis with the randomized data and checked the appropriate random results as well to be able to assess the significance of our results.

4 Results

4.1 Examined Software Systems

For the data analysis we used one industrial and three open-source software systems. For the industrial one we had all the information from the very beginning. For most of the open-source projects this is not the case; generally the initial source was merged from another version control system.

In order to gain as adequate results as possible, we considered only those projects for which we had at least 1,000 commits affecting at least one Java file. Furthermore, the too small code increase could also have significant bias, therefore we considered only those systems where the ratio of the maximal logical lines of code (typically the size of the system after the last available commit) and the minimal one (which was typically the size of the initial commit) was at least 3. We ended up with three such open-source systems.

Table 1 shows the basic properties of the systems on which the statistical analysis was performed. These are:

- **Gremon** – a greenhouse work-flow monitoring system.[2] It was developed by a local company between June 2011 and March 2012.
- **Ant** – a command line tool for building Java applications.[3]
- **Struts 2** – a framework for creating enterprise-ready java web applications.[4]
- **Tomcat** – an implementation of the Java Servlet and Java Server Pages technologies.[5]

Table 1: Analyzed systems

Name	Min.	Max.	Total	Java	Total number of			Rev. with 1+			Rev. with only		
	TLLOC[6]		Commits		A	U	D	A	U	D	A	U	D
Gremon	23	55,282	1,653	1,158	1,071	4,034	230	304	1,101	89	42	829	8
Ant	2,887	106,413	6,118	6,102	1,062	20,000	204	488	5,878	55	196	5,585	19
Struts 2	39,871	152,081	2,132	1,452	1,273	4,734	308	219	1,386	94	41	1,201	12
Tomcat	13,387	46,606	1,330	1,292	797	3,807	485	104	1,236	77	32	1,141	23

The first commit of the open-source projects started with a great amount of addition. In order to neutralize this bias we defined the quality change of the first commit to be 0.0. In case of Gremon, all the commits were analyzed from the very beginning to the very end.

[2] http://www.gremonsystems.com
[3] http://ant.apache.org
[4] http://struts.apache.org/2.x
[5] http://tomcat.apache.org
[6] Total Logical Lines Of Code – Number of non-comment and non-empty lines of code

4.2 The Input Contingency Tables

The contingency tables created for the examined projects can be found in Tables 2, 3, 4 and 5.

Table 2: Gremon

	A	D	$U+$	$U1$	Σ
+	118	43	122	54	337
0	13	3	126	223	365
-	109	43	198	106	456
Σ	240	89	446	383	1158

Table 3: Ant

	A	D	$U+$	$U1$	Σ
+	277	18	472	715	1482
0	13	12	625	2401	3051
-	172	25	467	905	1569
Σ	462	55	1564	4021	6102

Table 4: Struts 2

	A	D	$U+$	$U1$	Σ
+	123	43	183	149	498
0	17	25	166	503	711
-	82	46	233	179	540
Σ	222	114	582	831	1749

Table 5: Tomcat

	A	D	$U+$	$U1$	Σ
+	39	31	91	108	269
0	8	14	159	523	704
-	27	32	100	160	319
Σ	74	77	350	791	1292

There are a couple of notable facts about the tables. First of all, the distributions of the positive, neutral and negative commits within each commit category are different. Second, these distributions seem to be similar in every project. This is promising, and worth the effort of the detailed analysis.

A graphical overview of the data is shown in Figure 5, where the proportions of each commit category are illustrated on bar plot diagrams. The bars with different colors indicate the proportions of the positive (light gray), neutral (gray) and negative commits (dark gray) for each category, and the overall proportion is also displayed. In order to see the differences between the random and the actual data, the results of random executions for each project is also included (see Figure 6).

The following can be seen on these diagrams:

- The middle bars (gray) are smaller than expected in case of A, D and U+, and higher in case of U1.
- The upper bar (light gray) is the tallest in case of A on every diagram.
- In case of U+ and U1 the lower bars (dark gray) are bigger than the upper ones (light gray) in most of the cases.

The relevance of these results are very spectacular if we compare them to the bar plots of the randomized data (see Figure 6). In case of randomized data, there are no obvious differences in any category bar, compared to the bar of all commits (or with the bar of any other category). Furthermore, even the viewable small differences in the bars do not tend to be relevant: one difference on one diagram mostly differs on the other ones.

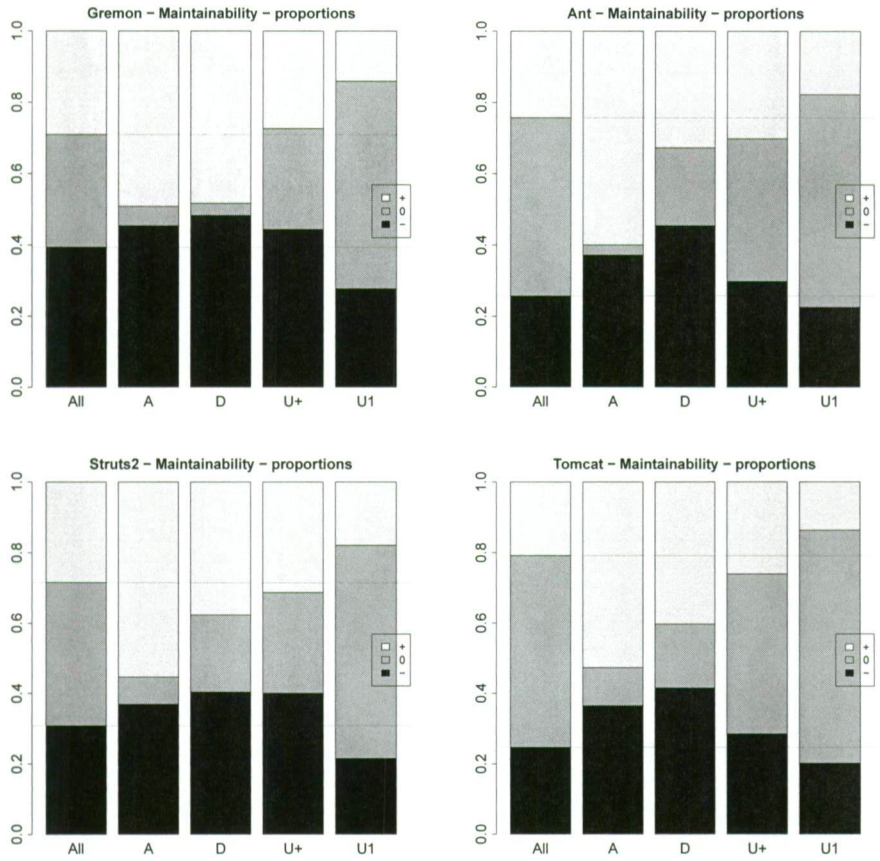


Figure 5: Maintainability proportions

4.3 Results of Contingency Chi-Squared Tests

Based on the bar plot diagrams (see Figure 5) we already have an assumption about the answers to our research questions, but for a more grounded answer let us check the results of the Chi-squared tests on the contingency tables. In case of the Gremon project we present all the details. For the open-source systems only the input and the final results are shown from which the main conclusions can be drawn.

As already mentioned, Table 2 presents the original contingency table for the Gremon project on which the test was performed on. For example, the meaning of the upper left value (118) is the following: the total number of commits containing no deletion, containing at least one addition (i.e. belongs to category A based on the definition) and the maintainability change caused by that commit was positive. The last row and the last column contains the sum of the values of the appropriate

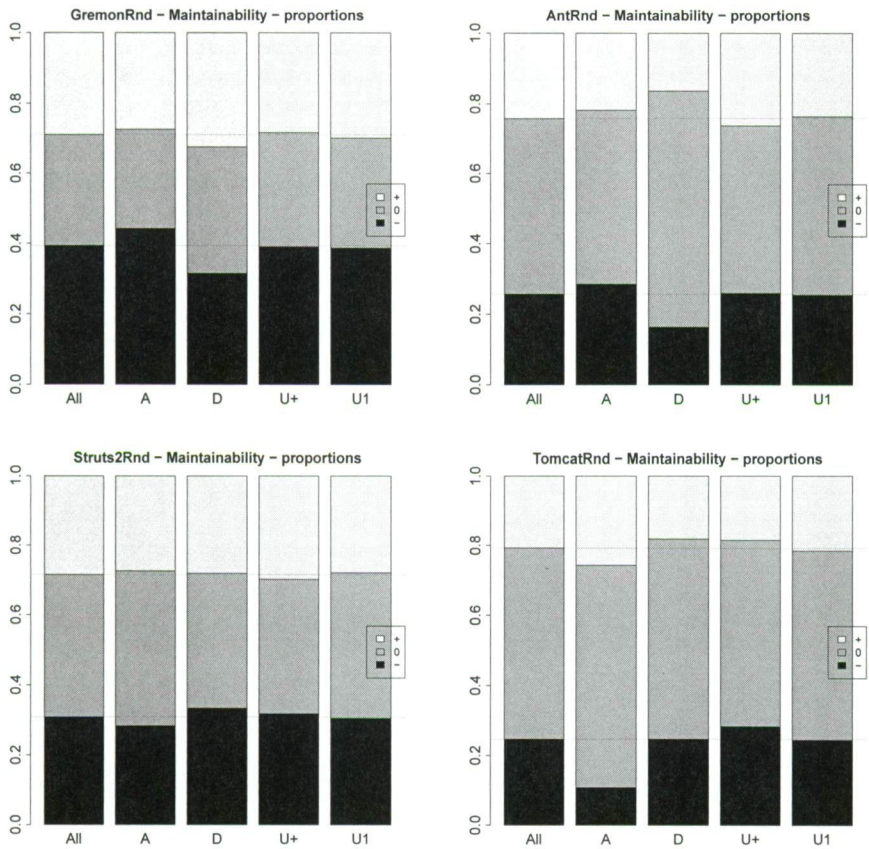


Figure 6: Maintainability proportions in random cases

rows and columns, respectively. This is the consolidated input of the contingency Chi-squared test.

Table 6 contains the calculated expected values. Practically, this is the null-hypothesis: if the row and column sums would be the same as in the case of measured data, then in case of uniform distribution these were the cell values. The average values of random cases would tend to this matrix. The sums of rows and columns are the same as in the previous table. The meaning of the upper left value (69.8) is the following: if there was no connection between version control operations and maintainability change, and the number of commits in each category would be the same as in case of the input, furthermore, the total numbers of positive, neutral and negative maintainability changes were also the same, then this value would be an integer close to this number. In other words: the average value of this cell in the random cases would tend to this value. In this case the value 69.8 is much smaller than the value 118 found in the previous matrix (see Table 2).

Table 7 shows the standard residuals. This table illustrates if the previous difference is significant or not using the well-known standard normal distribution. The difference between the expected and the measured value is exactly as extreme as the difference between 0 and the values found in this table assuming a standard normal distribution. E.g., in the upper left case this is the chance of resulting in 7.69. Based on this, we already have a feeling that this is a very extreme value; the probability of resulting such value only by chance is very low.

Table 6: Gremon: expected values

	A	D	$U+$	$U1$	Σ
+	69.8	25.9	129.8	111.5	337
0	75.6	28.1	140.6	120.7	365
-	94.5	35.0	175.6	150.8	456
Σ	240	89	446	383	1158

Table 7: Gremon: standard residuals

	A	D	$U+$	$U1$
+	7.69	4.15	-1.04	-7.90
0	-9.78	-5.95	-1.89	13.75
-	-2.15	1.80	2.77	-5.73

In Table 8 we present the p-values related to the standard normal distribution. These values answer the question of how low the previously mentioned chances are. Consider the upper left value again. The difference between the actual value (118) and the expected value (69.8) is 48.2. The other value with the same difference from the expected one is 21.6 (=69.8-48.2). The definition of the p-value is the following: the chance of the value being at least as extreme as measured, provided that the null-hypothesis is true. Therefore the meaning of the value in the upper left corner ($1.52 \cdot 10^{-14}$) is the following: the chance that the measured value is at least 118 or at most 21.6. Taking into consideration that its reciprocal is about $6.58 \cdot 10^{13}$ it means that in random case this would statistically happen once in about every 66 trillion cases (and about once in every 132 trillion cases if the direction also matters).

Table 8: Gremon: p-values

	A	D	$U+$	$U1$
+	$1.52 \cdot 10^{-14}$	$3.28 \cdot 10^{-5}$	$3.00 \cdot 10^{-1}$	$2.76 \cdot 10^{-15}$
0	$1.43 \cdot 10^{-22}$	$2.70 \cdot 10^{-9}$	$5.81 \cdot 10^{-2}$	$5.06 \cdot 10^{-43}$
-	$3.15 \cdot 10^{-2}$	$7.25 \cdot 10^{-2}$	$5.69 \cdot 10^{-3}$	$1.01 \cdot 10^{-8}$

Table 9 contains the exponents calculated as described in Section 3.5. Theoretically, the previous tables contain everything we need: the standard residuals provide the directions and the p-values table provide the absolute values; but the tables containing the exponents are easier to read and comprehend.

Table 9 is composed of the exponents and the directions. Consider the upper left value (13). The absolute value comes from the exponent (14) minus one (in order to convert the absolutely not significant results (having p-value > 0.1) to 0 instead of 1). The sign means the direction: the positive in this case means that the actual value is higher than the expected one. Also note that although this value is high, it is still far from the highest.

Tables 10, 11 and 12 show the resulted exponents for the Ant, Struts 2 and Tomcat projects, respectively.

Table 9: Gremon: exponents

	A	*D*	*U+*	*U1*
+	13	4	0	-14
0	-22	-8	-1	42
-	-1	1	2	-7

Table 10: Ant: exponents

	A	*D*	*U+*	*U1*
+	76	1	9	-60
0	-98	-4	-19	98
-	8	3	4	-14

Table 11: Struts 2: exponents

	A	*D*	*U+*	*U1*
+	20	1	1	-19
0	-26	-4	-12	57
-	1	1	8	-15

Table 12: Tomcat: exponents

	A	*D*	*U+*	*U1*
+	11	4	2	-14
0	-14	-10	-4	25
-	1	3	1	-5

Table 13 summarizes the overall p-values of each contingency Chi-Squared test (the previous p-values are calculated on a per cell basis).

Table 13: Overall p-values

Project	**p-value**
Gremon	$1.19 \cdot 10^{-52}$
Ant	$1.60 \cdot 10^{-151}$
Struts 2	$4.47 \cdot 10^{-64}$
Tomcat	$4.84 \cdot 10^{-33}$

Based on these extremely low overall p-values in every case, we can state that there is a strong connection between the version control operations and the maintainability changes.

For getting a better overview, the resulted exponents are summed up and presented in Table 14, indicating those cells where the results are significantly similar for the systems. Dark cell means that the absolute value in every case was at least 2 ($p < 0.01$). The darkness indicates the degree of similarities in the significance. If there are 2 or 3 significant results and 1 not significant, it is indicated with a lighter color. 0 or 1 significant result is denoted by an even lighter cell fill. White is reserved for significant contradictions, i.e. if a cell would contain -2 or less in one case, and +2 or more in the other.

Table 14: Overview: sum of the exponents

	A	*D*	*U+*	*U1*
+	120	10	12	-107
0	-160	-26	-36	222
-	9	8	15	-41

Half of the cells are dark; these indicate those results which are significant for every checked project. Please note that the table does not contain any white cells.

4.4 Random Check Result

We were also interested in the random case: does it also result in the same high numbers as presented previously or not. Based on the definition of the exponent table, theoretically, in random case the proportion of 0 should be 90%, the proportion of absolute values 1 should be 9% (half of them negative and half of them positive), the proportion of absolute values 2 should be 0.9%, and so on. We received approximately the same kind of distributions in practice. Table 15 illustrates the results of one concrete execution with an overall p-value of 0.53. There are hardly any non-null values in these executions.

Table 15: Random: exponents

	A	D	$U+$	$U1$
+	0	0	0	0
0	0	0	0	0
-	1	0	0	0

4.5 Answers to the Research Questions

The answers to the research questions are primarily based on Table 14.

RQ1: *Do commits containing file additions to the system have a significant positive impact on its maintainability?*

The values in the first column are related to these commits. Value 120 and the dark color cell in the upper left cell indicates that the positive impact on the maintainability is very high for those commits which do not contain deletion but contain at least one addition. *This supports our assumption that adding new source files to the system has a significant positive impact on its maintainability.*

On the other hand, the lower left cell of the table is also positive (+9), but the color is lighter. In 3 out of the 4 cases it contained a value close to 0, and one higher value. If we check the input, we see that the absolute number of commits in the positive cell is also higher than those in the negative cell in every case. Therefore we can also say that the overall effect of the add operation is positive.

RQ2: *Is it true that the commits containing source file updates only tend to significantly decrease the maintainability?*

The third and fourth columns are related to commits containing updates only. All the colors of the cells found in the fourth column (commits containing exactly one update) are dark and the values are negative both in the + and - cells. But the value found in the + row is much lower than the value found in the - row, and this is true for every input. We should also take into account that the maintainability tends to decrease, therefore if these values were equal, that would also

mean maintainability decrease as an overall result. Thus *in case of commits containing one update our assumption that the source file updates tend to decrease the maintainability is supported with high significance.*

The cell colors in the third column (commits containing exclusively at least 2 updates) are lighter. Both of the values found in the + and - rows are positive. However, the value found in the - cell is higher than the value in the + cell. Therefore *in case of more updates our assumption is also supported, but with lower significance.*

RQ3: *Do commits containing file deletion improve the maintainability of the system?*

The second column is related to this research question. The values found in these cells are small in absolute values compared to those found in other columns and their colors are also not the darkest ones. The number in the + cell (10) is higher than the number in the - cell (8). Based on this we cannot formulate a general statement. Seemingly we could say that deletions have no positive effect on the maintainability as $10 > 8$. But that could be a false conclusion, because in general the number of commits causing negative maintainability change is in general higher than those causing positive change. Therefore 10 in the + cell does not necessarily mean higher number of absolute values than 8 in the - cell. And if we check the inputs, we see that just the opposite is true, i.e. the absolute number in the - cells in columns D are less than or equal with the values in the + cells. If we consider the input as well we find that there are more such commits of category D which resulted in maintainability decrease than those of increase. Therefore *the third assumption that commits containing deletion improve the maintainability of the system is not supported by the results.*

4.6 Other Results

Considering Table 14 other results can also be read out, not covered by the original research questions.

First of all, the highest absolute value is 222, in row 0, column U1. All the other values in row 0 are negative. This means that *no traceable maintainability changes are primary related to small updates.* This is a trivial statement, of course, and it rather validates the used quality model than a real usable result of this research.

The second highest value in absolute is -160, also in row 0, but column A. Therefore *adding a new source code almost always has some traceable effect on the maintainability.*

Considering the negative (-) row alone, it would lead the false result that every commit category have negative effect on the maintainability, except the small updates. This is not true, because the value found in the positive (+) row should also be considered in case of every category. On the other hand, these values tell us that *with the exception of small updates there are too many maintainability decreases.* Eliminating some of these decreases would result in a well maintainable code, even without an explicit code quality increase campaign.

5 Threats to Validity

In some cases we achieved very convincing results; however, there are some facts that may threat the validity of some of them.

In case of the open-source projects, the first unknown number of commits are missing (most probably they were migrated from another version control system). On the other hand, in the case of Gremon, all the commits were available from the very beginning. This inconsistency may lead to false results in some cases; however, it would be interesting to investigate the differences between the commits in the beginning and commits in a later phase of the development. For that, a much greater amount of data would be necessary.

There are a few diverging results: in most of these cases there are two or three similar results, but the other system(s) do(es) not support that. In general it does not contradict them either so it does not mean necessarily that the results are invalid. This may be caused by several issues: the divergence may be caused by the domain differences, technological differences, differences in development processes, the different phases examined, or simply that there are maybe exceptions under the general rule, and some of the examined systems fall into these exceptions. This is by all means worth further investigations.

6 Conclusions and Future Work

In this work we studied the impact of version control commit operations on maintainability change. We found that commits containing Update operation only have negative impact on maintainability, while great maintainability improvements are mostly caused by those commits which contain Add operations as well. Commits containing operation Delete have a weak connection with maintainability, and we could not formulate a general statement. Operation Rename was not investigated on its own due to the very small number of its occurrences and due to the fact that this on its own does not have any measurable effect on the maintainability.

Another conclusion is that commits consisting of a single Update tend to have no traceable impact on maintainability. On the other hand, other types of commits tend to have significant impact on maintainability.

Based on these results it might make sense for developers to improve the way they add new features and use the opportunity to also perform refactorings. The new features should be implemented in new files, containing sound code (adding new files typically improve maintainability), and the existing code should be refactored to accept the new code in the proper way (refactorings typically introduce file deletions and additions). Modifications are of course more expensive in this way, but the extra investment returns in mid-term.

During the analysis we used only a subset of the available data. Extending the analysis with other types of commit-related data, like the file name, the date and time of the commit, the developer, or the comment belongs to our short-term plans. As already mentioned among the threats to validity, we did not take into

consideration the domain and other attributes of the software. That could be an important extension of this work for a mid-term future research. In longer term, we plan to include non-version control related data into consideration as well. For instance, useful information may be extracted from the issue tracking systems. Finally, we have a great expectation from the results of those tests where the developer interactions collected by the IDE are also considered.

Acknowledgments

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0006, and the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4. A/2-11-1-2012-0001 „National Excellence Program".

References

[1] Agresti, Alan. *An Introduction to Categorical Data Analysis.* Wiley-Interscience, 2 edition, March 2007.

[2] Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., and Gyimóthy, T. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*, pages 368–377, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[3] Bakota, Tibor, Hegedűs, Péter, Ladányi, Gergely, Körtvélyesi, Péter, Ferenc, Rudolf, and Gyimóthy, Tibor. A Cost Model Based on Software Maintainability. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pages 316–325, Riva del Garda, Italy, 2012. IEEE Computer Society.

[4] Bernstein, A and Bachmann, A. When Process Data Quality Affects the Number of Bugs: Correlations in Software Engineering Datasets. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, MSR '10*, pages 62–71, 2010.

[5] Giger, Emanuel, Pinzger, Martin, and Gall, Harald C. Can We Predict Types of Code Changes? An Empirical Analysis. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 217–226. IEEE, 2012.

[6] Hegedűs, Péter. A Probabilistic Quality Model for C# – an Industrial Case Study. *Acta Cybernetica*, 21(1):135–147, 2013.

[7] Hindle, Abram, German, Daniel M., and Holt, Ric. What Do Large Commits Tell Us?: a Taxonomical Study of Large Commits. In *Proceedings of the 2008*

International Working Conference on Mining Software Repositories, MSR '08, pages 99–108, New York, NY, USA, 2008. ACM.

[8] ISO/IEC. *ISO/IEC 9126. Software Engineering - Product quality 6.5.* ISO/IEC, 2001.

[9] Khoshgoftaar, Taghi M., Allen, Edward B., Halstead, Robert, Trio, Gary P., and Flass, Ronald M. Using Process History to Predict Software Quality. *Computer*, 31(4):66–72, April 1998.

[10] Koch, S. and Neumann, C. Exploring the Effects of Process Characteristics on Product Quality in Open Source Software Development. *Journal of Database Management*, 19(2):31, 2008.

[11] Moser, Raimund, Pedrycz, Witold, Sillitti, Alberto, and Succi, Giancarlo. A Model to Identify Refactoring Effort during Maintenance by Mining Source Code Repositories. In *Proceedings of the 9th International Conference on Product-Focused Software Process Improvement*, PROFES '08, pages 360–370, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 181–190, New York, NY, USA, 2008. ACM.

[13] Nagappan, Nachiappan, Ball, Thomas, and Murphy, Brendan. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE '06)*, pages 62–74, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Nagappan, Nachiappan, Ball, Thomas, and Zeller, Andreas. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 452–461, New York, NY, USA, 2006. ACM.

[15] Parnas, David Lorge. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[16] Peters, Ralph and Zaidman, Andy. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 411–416, Washington, DC, USA, 2012. IEEE Computer Society.

[17] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013.

[18] Ratzinger, Jacek, Sigmund, Thomas, Vorburger, Peter, and Gall, Harald. Mining Software Evolution to Predict Refactoring. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 354–363, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Schofield, Curtis, Tansey, Brendan, Xing, Zhenchang, and Stroulia, Eleni. Digging the Development Dust for Refactorings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 23–34, Washington, DC, USA, 2006. IEEE Computer Society.

[20] van Rysselberghe, F. and Demeyer, S. Mining Version Control Systems for FACs (frequently Applied changes). In *Proceedings of the International Workshop on Mining Repositories*, Edinburgh, Scotland, UK, 2004.

[21] Ying, Annie T. T., Murphy, Gail C., Ng, Raymond, and Chu-Carroll, Mark C. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[22] Ying, Annie T. T. and Robillard, Martin P. The Influence of the Task on Programmer Behaviour. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 31–40, Washington, DC, USA, 2011. IEEE Computer Society.

[23] Zimmermann, Thomas, Weissgerber, Peter, Diehl, Stephan, and Zeller, Andreas. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Received 20th November 2013

Acta Cybernetica **21** (2014) 609–628.

Computing Equivalent Affinity Classes in a Fuzzy Connectedness Framework

Gergely Gulyás* and József Dombi†

Abstract

The equivalence of affinities in fuzzy connectedness (FC) is a novel concept which gives us the ability to study affinity functions and their precise connection with FC algorithms. Two seminal papers by Ciesielski and Udupa create a strong theoretical background and provide some useful practical examples. Our intention here is to investigate this concept further because from a practical viewpoint if we are able to determine the equivalence classes for a given set of affinity functions and narrow it down to a much smaller set of nonequivalent affinities, then the set can be used more effectively in an optimization framework which searches for the best affinity function or parameters for a special task. In other words, we can find the best configuration for a set of given hardware or an image set with special characteristics. From a theoretical perspective, we are interested in the complexity of this problem, i.e. determining equivalence classes. Here, an affinity operator is used which is a function of a given parameter and maps different parameter values for different affinity functions. Our first questions, namely how many different meaningful, non-equivalent affinities there are and how we can enumerate them, led us to a general problem of how the equivalent affinities partition the parameter's domain and how the corresponding equivalence classes can be determined. We will provide a general algorithm schema to construct special algorithms which are able to compute the equivalence classes. We will also analyze a special but very common scenario of when the affinity operator combines two affinities (e.g. a homogeneity and an object feature-based affinity) using an aggregation operator (e.g. weighted average) and the particular parameter defines the weights of the affinities. Based on the general algorithm schema, we propose algorithms for this special case and we determine their complexity as well. These algorithms will be tested on two sets of medical images, namely, 25 digital dermoscopy images 1280×1024 pixels in size and 3×25 simulated brain MRI slices 181×217 in size.

Keywords: Fuzzy Connectedness, Affinity Functions, Equivalence of Affinities, Image Segmentation, Equivalence of algorithms

*E-mail: gulyasg@inf.u-szeged.hu, gergely.gulyas.uni@gmail.com
†E-mail: dombi@inf.u-szeged.hu

1 Introduction

In general, the goal of image segmentation is to partition the image into meaningful object regions. However, this is one of the most difficult tasks in the image processing realm with numerous open questions. Many different approaches exist to handle this problem, one of the most popular being the family of region-based algorithms [3, 4, 32, 34] where the objects are described by their filled regions.

Fuzzy techniques are also widely used in image processing [5, 6, 7, 19, 26, 33], due to the fact that they address the problem of ambiguity in digital images (caused by noise or imprecision, for example). Fuzzy connectedness (FC) [12, 23, 29, 30, 34, 35, 38] is a region-based, fuzzy segmentation framework that has good theoretical support and has been used successfully in several medical applications [21, 22, 28, 31, 36].

In the FC framework, a global fuzzy relation, called fuzzy connectedness, characterizes how the image elements hang together to build up objects. The strength of this relation between any two image elements (or spels) c and d, which refers to the strength of their connectedness, can be determined in the following way. Consider all the possible paths connecting c and d. Each path is a sequence of spels, starting from c and ending in d, with the successive spels being nearby. Each consecutive spel pair constitutes a link and we assign a strength to every path, which is the strength of the weakest link along the path. The strength of connectedness between c and d is the strength of the strongest path between c and d. A local fuzzy relation, called the affinity function, is used to determine the strength of the consecutive spel pairs (i.e. the strength of links). Generally speaking, the strength of affinity between any two spels depends on how close the spels are spatially and how similar their properties (like intensity and colour) are in the image.

FC algorithms are parametric in nature. This means that object feature-based affinities require some a priori knowledge about the object (e.g. expected mean and standard deviation of the intensities [23]), while object feature-based and homogeneity affinities are combined in many scenarios where the aggregation operator requires some weights [35]. One of the most challenging problems with parametric algorithms in real applications is to find the optimal parameter values, because a given solution can only be evaluated by human observation and cannot be automated. In addition, the parameter domain (i.e. the search space, which may be large or infinite) and the running time of the particular algorithm can also limit the speed of testing. Equivalence of affinities [10, 11], which is a novel concept in the FC framework, gives us the ability to address this problem. Informally, if two affinities used in the same FC schema are equivalent, they lead to identical segmentations [10]. Accordingly, if we could filter the redundant, equivalent affinities from our experimental set, it would definitely cut down the search space. Here, we focus on the theoretical background and algorithmic questions of the former, namely how many different meaningful, non-equivalent affinities there are and how we can enumerate them; or more generally, how the equivalence classes can be characterized. In our model scenario, we have an affinity operator (Sec. 2.2) that is a function of a given parameter, so it maps different parameter values for different affinities. For

example, suppose we have two affinities κ_1 and κ_2 and we combine them into κ_w using the weighted average operator with the parameter w:

$$\kappa_w = w\kappa_1 + (1 - w)\kappa_2.$$

In this study, we propose using a general algorithm schema to create special algorithms which are able to determine the set of equivalence classes based on the parameter value of the affinity operator. Then, we investigate a very common scenario where two affinities are combined by means of a weighted quasi-linear mean [18] (e.g. weighted average as in the example above) and some concrete algorithms are built based on the general algorithm schema. The complexity of these algorithms are also considered, and we will show that the structure of equivalence classes is very simple in this case. Lastly, we test the algorithms on medical image sets where our goal is to see how many different equivalence classes (i.e. non-redundant affinities) belong to a given image; in other words how big the search space is in the case of real applications.

2 Equivalence of affinities

Now, we will briefly present some standard concepts and definitions that will be used throughout, which are well known in fuzzy theory and more detailed descriptions can be found in the literature [10, 34, 35].

2.1 Basic notations and definitions

Let $\mathbb{Z}^n$ stand for the set of all n-tuples of integers. A binary fuzzy relation α on $\mathbb{Z}^n$ ($n \geq 2$) is a *fuzzy adjacency* if α is symmetric and reflexive. The pair $\langle \mathbb{Z}^n, \alpha \rangle$ is called an n-dimensional *fuzzy digital space.* A *scene* over a fuzzy digital space $\langle \mathbb{Z}^n, \alpha \rangle$ is a pair $\mathcal{C} = \langle C, f \rangle$, where $C = \prod_{j=1}^{n} [-b_j, b_j] \subset \mathbb{Z}^n$, each $b_j > 0$ being an integer, and $f\colon C \to \mathbb{R}^k$ is a scene *intensity function* ($k \geq 1$). If the range of f is a subset of the interval $[0, 1]$, the scene is called the *membership scene.*

2.2 Affinity functions

Affinity is a binary fuzzy relation which indicates how two spels hang together locally in the scene, its strength depending on how close these spels are spatially and how similar their properties are in the image. It plays a crucial role in the FC framework because the global fuzzy connectedness of spels is derived by means of their affinities. The following definition gives a general characterization of affinity functions [10].

Definition 1. *Let $\preceq$ be a linear order relation [13] on a set L and let C be an arbitrary finite non-empty set. A function $\kappa\colon C \times C \to \langle L, \preceq \rangle$ is an* affinity function *from C into $\langle L, \preceq \rangle$ if κ is symmetric and $\kappa(a, b) \preceq \kappa(c, c)$ for every $a, b, c \in C$.*

We say that κ is a *standard affinity* if it is a function taken from C to $\langle [0,1], \leq \rangle$. In practice, the value of $\kappa(c,d)$ depends on the adjacency strength $\alpha(c,d)$ of c and d and on the intensity function f. In our experiments, we use the following two well-known types of affinities taken from the literature [11, 30].

Definition 2. *Let $\psi\colon C \times C \to \langle [0,1], \leq \rangle$ be a standard homogeneity-based affinity function such that for every $c, d \in C$*

$$\psi(c,d) = \begin{cases} 1 & \text{if } c = d \\ e^{-|f(c)-f(d)|^2} & \text{if } \|c-d\| = 1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Definition 3. *Let $\phi\colon C \times C \to \langle [0,1], \leq \rangle$ denote a standard object feature-based affinity such that for every $c, d \in C$*

$$\phi(c,d) = \begin{cases} 1 & \text{if } c = d \\ min(e^{-|f(c)-m|^2/\sigma^2}, e^{-|f(d)-m|^2/\sigma^2}) & \|c-d\| = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

where m and σ are the expected mean and standard deviation values of object intensities.

We should remark that in Def. 3 a family of affinities are defined where affinities differ in parameter values m and σ (our expectations or a prior knowledge about the object). These values are usually associated with a given scene and therefore this association can be treated as an *affinity operator*

$$\mathcal{K}(\mathcal{C}, m, \sigma) := \langle \mathcal{C}, m, \sigma \rangle \mapsto \kappa_{m,\sigma},$$

which produces affinity functions based on its input parameters.

2.3 Fuzzy connectedness schemas

Next, we will briefly describe the concept of fuzzy connectedness and the algorithm schemas, but we will avoid any formal definitions and theorems because these can be found in the literature cited and are not too important in this study (due to the fact the concept of equivalent affinities is valid in all of these schemas).

Fuzzy connectedness is a binary fuzzy relation which refers to the global hanging togetherness of the spel pairs in a scene as follows. For a given spel pair c and d, we consider all possible connecting paths between them and the level of connectedness is defined by the maximum of the strengths of all paths. The strength of a path is the minimum of the affinities of consecutive spels along the path [34]. Intuitively, the higher the level of connectedness between to spels, the higher the probability that these spels belong to the same object.

There are some well-known and commonly used algorithm schemas (or FC frameworks) which are able to determine different segmentations for a given scene based on some affinity functions. These are absolute FC (AFC) [35], relative FC (RFC) [29], iterative RFC (IRFC) [12], scale-based [30] and vectorial FC [38].

2.4 Equivalent affinities

Equivalence of affinities [10, 11] is a key notion in our study. Informally, two affinity functions are equivalent in the FC sense if they lead to identical segmentations when applied to any scene starting from the same seeds. The following definition [10] characterizes this concept more formally, which constitutes the basis for our study here.

Definition 4. *The affinities* $\kappa_1 \colon C \times C \to \langle L_1, \preceq_1 \rangle$ *and* $\kappa_2 \colon C \times C \to \langle L_2, \preceq_2 \rangle$ *are equivalent in the FC sense if for every* $a, b, c, d \in C$

$$\kappa_1(a,b) \preceq_1 \kappa_1(c,d) \iff \kappa_2(a,b) \preceq_2 \kappa_2(c,d). \tag{3}$$

The statement (i.e. two equivalent affinities, as described in Def. 4 lead to identical segmentations), is presented and proven in [10] (Theorem 5).

Without loss of generality, we shall restrict our investigation to standard affinities due to the following theorem [10].

Theorem 1. *Every affinity function is equivalent in the FC sense to a standard affinity.*

Proof. See proof in [10]. □

3 General algorithm schema

One of the chief goals of our study is to provide algorithms that are able to determine equivalence classes for a set of affinities. This is important from a practical viewpoint because we should avoid the use of equivalent affinities during a real application. However, from a theoretical perspective, this provides the basis for investigating equivalence classes, like the number of classes compared to the cardinality of the affinity set. Here, we will present a general algorithm schema (mostly based on Def. 4), which is the first step towards defining these algorithms.

In our example, we will assume that we have an affinity operator which depends on a real parameter w, i.e. it provides a set of affinities. We should add that the operator may also depend on a given scene $\mathcal{C}$ and on some additional parameters, but we view these as fixed parameters due to the affinity equivalence being restricted to a particular scene. We propose an algorithm schema which takes an affinity operator and a scene as inputs and determines the set of affinity equivalence classes. It is an abstract algorithm because it is to be implemented for a particular family of affinity operators, highlighting the tasks and providing an algorithm template for more specific algorithms. We will present some actual applications later on.

More formally, assume that a fixed scene $\mathcal{C}$ is given and there is an affinity operator

$$\mathcal{K}(\mathcal{C}, w, p) := \langle \mathcal{C}, w, p \rangle \mapsto \kappa_w,$$

where w is the given parameter such that

$$w \in [L, U] \subseteq \mathbb{R}$$

and p represents all other parameters (which are dependent on $\mathcal{C}$). A certain affinity function for a given w is referred to as κ_w because the scene and the other parameters are fixed (hence the indices can be omitted to).

The following structure (defined in Def. 5) is the key component here because it provides a formal description of the set of affinity equivalence classes.

Definition 5. *Let γ be an equivalence relation on the interval $[L, U]$ such that*

$$\gamma = \{\langle w_1, w_2\rangle \in [L, U] \times [L, U] : \kappa_{w_1} \text{ and } \kappa_{w_2} \text{ are equivalent in FC sense}\},$$

and let G denote the set of the equivalence classes induced by γ.

The following definitions are used to construct the algorithm schema which determines G (Def. 5) based on the definition of equivalent affinities (Def. 4). First, suppose that a certain 4-tuple of spels $\langle a, b, c, d\rangle$ is fixed $(a, b, c, d \in C)$.

Definition 6. *Let $\Delta\colon [L, U] \to \{-1, 0, 1\}$ be a function such that*

$$\Delta(w) = sgn(\kappa_w(a, b) - \kappa_w(c, d)),$$

where sgn denotes the sign function.

Obviously, if $\Delta(w_1) = \Delta(w_2)$ then the corresponding κ_{w_1} and κ_{w_2} define the same ordering on the spel pairs (a, b) and (c, d).

Definition 7. *Let ρ be an equivalence relation on the interval $[L, U]$ such that*

$$\rho = \{\langle w_1, w_2\rangle \in [L, U] \times [L, U] : \Delta(w_1) = \Delta(w_2)\},$$

and let $P = \{P^{(-)}, P^{(0)}, P^{(+)}\}$ denote the set of the equivalence classes belonging to ρ, where

$$\begin{aligned} P^{(-)} &= \{w \in [L, U] : \Delta(w) = -1\}, \\ P^{(0)} &= \{w \in [L, U] : \Delta(w) = 0\}, \\ P^{(+)} &= \{w \in [L, U] : \Delta(w) = 1\}. \end{aligned}$$

Thus, each w in $P^{(-)}$ satisfies $\kappa_w(a, b) < \kappa_w(c, d)$. The sets G and P are partitions of $[L, U]$.

The general algorithm schema which computes G can be seen in Alg. 1. The procedure starts with an initial partition (step 1) which contains the interval $[L, U]$ itself. Then, it iterates over the possible 4-tuples (steps 2 - 9) and determines the set of equivalence classes P for each 4-tuple (Step 3). In steps 4-8, the algorithm refines the current partition G with the elements of P, i.e. if P_{curr} and an element G_{curr} of G intersect, then the algorithm replaces G_{curr} by the intersection and the difference. The purpose of this step is to merge the different partitions of the 4-tuples into a global partition which describes γ and G (more formally, in Prop. 1). The performance of the partition refinement step depends on the general structure of the different partitions, and many algorithms and data structures used to implement this step can be found in the literature [16, 24, 27].

Algorithm 1 General algorithm schema

Input: the operator $\mathcal{K}$, the fixed $\mathcal{C}$, p and the domain $[L, U] \subseteq \mathbb{R}$ of w
Output: G

```
1: G ← {[L, U]}
2: for all 4-tuple ⟨a, b, c, d⟩ do
3:    determine P = {P^(−), P^(0), P^(+)} (according to Δ)
4:    for all P_curr ∈ P and G_curr ∈ G do
5:       if P_curr ∩ G_curr ≠ ∅ then
6:          substitute G_curr in G by G_curr ∩ P_curr and G_curr \ P_curr
7:       end if
8:    end for
9: end for
10: return G
```

Proposition 1. *The Alg. 1 computes G correctly.*

Proof. Suppose that the algorithm iterates over all possible 4-tuples in a given $t_1, t_2, \ldots, t_k$ order (where $t_i = \langle a, b, c, d \rangle$ is a 4-tuple of spels), and let P_i denote the partition P belonging to t_i and $G^{(i)}$ the state of G in the ith iteration. We would like to prove that in the ith iteration, $G^{(i)}$ consist of the equivalence classes belonging to the first i 4-tuples $(t_1, \ldots, t_i)$, i.e. γ is correct if its verification is restricted to these 4-tuples.

In the first step, when $i = 1$, the initial $[L, U]$ is partitioned by P_1, which means that $G^{(1)} = P_1$. Now, suppose that the statement is satisfied for i, thus γ is correct when restricted to $t_1, \ldots, t_i$. Then the method takes t_{i+1} and its partition $P_{i+1} = \{P_{i+1}^{(-)}, P_{i+1}^{(0)}, P_{i+1}^{(+)}\}$. Because P_{i+1} is a partition, each $G_{curr} \in G^{(i)}$ will be substituted by $P_{i+1}^{(-)} \cap G_{curr}$, $P_{i+1}^{(0)} \cap G_{curr}$ and $P_{i+1}^{(+)} \cap G_{curr}$, where at least one intersection is not empty.

Next, consider a non-empty intersection like $P_{i+1}^{(0)} \cap G_{curr}$, and let $w \in P_{i+1}^{(0)} \cap G_{curr}$. The affinities belonging to this set are equivalent in the FC sense restricted to $t_1, \ldots, t_i, t_{i+1}$, because for each parameter $w' \in P_{i+1}^{(0)} \cap G_{curr}$, the corresponding κ_w and $\kappa_{w'}$ define the same ordering on the spel pairs of the 4-tuples $t_1, \ldots, t_i, t_{i+1}$ due to the definition of G_{curr} $(t_1, \ldots, t_i)$ and due to the definition of $P_{i+1}^{(0)}$ (t_{i+1}). For each $w'' \in G_{curr} \setminus P_{i+1}^{(0)}$, κ_w and $\kappa_{w''}$ define a different ordering on t_{i+1}, and for each $w''' \in P_{i+1}^{(0)} \setminus G_{curr}$, κ_w and $\kappa_{w'''}$ define a different ordering at least once on the 4-tuples $t_1, \ldots, t_i$. Thus, $P_{i+1}^{(0)} \cap G_{curr}$ is an equivalence class in the FC sense restricted to $t_1, \ldots, t_i, t_{i+1}$. The proof is similar to $P_{i+1}^{(-)} \cap G_{curr}$ and $P_{i+1}^{(+)} \cap G_{curr}$. So the algorithm replaces all the subsets of $G^{(i)}$ by equivalence classes (restricted to the first $i + 1$ 4-tuples); and the induction step is satisfied. □

4 Aggregating two affinities by weighted quasi-linear means

Next, we will investigate a more specific scenario, when a particular affinity combines two other affinities (e.g. homogeneity and object feature-based affinities) by means of an aggregation operator. These affinity functions are often used in real applications and as examples in the literature [7, 11, 23, 30, 34, 35]. Thus, in this example, our affinity operator depends on two affinity functions and an aggregation operator with a weight parameter w. The authors in [11] discuss the problem of combining affinities, and they use a weighted arithmetic mean, weighted geometric mean, and lexicographical order to aggregate affinity functions (other work on this topic can be found in [25]). Here, we study the first two, more precisely their general class i.e. quasi linear means, and we investigate the structure of equivalence classes and introduce several implementations of the general algorithm schema (Alg. 1) for this particular case.

4.1 The structure of equivalence classes regarding 4-tuple of spels

A characterization of quasi linear means can be found in Theorem 2 [18]. We should add that this class of mean operators involves the weighted forms of arithmetic, geometric, harmonic and root-power means.

Theorem 2. *An operator $M^{(m)}$ is continuous, strictly monotonic, idempotent and bisymmetrical if and only if $M^{(m)}$ represents a quasi-linear mean, i.e.*

$$M^{(m)}(x_1,\ldots,x_m)=\varphi^{-1}\left(\sum_{i=1,\ldots,m}\omega_i\varphi(x_i)\right),\quad \omega_i\geq 0,\quad \sum_{i=1,\ldots,m}\omega_i=1,$$

where $\varphi\colon [0,1]\to[0,1]$ is an increasing continuous function.

Proof. See [2, 1]. □

Definition 8. *Suppose that $\mathcal{C}=\langle C,f\rangle$ is a scene, $a,b\in C$, $\varphi\colon[0,1]\to[0,1]$ is a continuous increasing function, $\kappa_1,\kappa_2\colon C\times C\to\langle[0,1],\leq\rangle$ are standard affinity functions, and let $\mathcal{K}$ be an affinity operator such that*

$$\mathcal{K}(w,\mathcal{C},\kappa_1,\kappa_2):=\langle w,\mathcal{C},\kappa_1,\kappa_2\rangle\mapsto\kappa_w,$$

where $w\in[0,1]$ and $\kappa_w\colon C\times C\to\langle[0,1],\leq\rangle$ such that

$$\kappa_w(a,b)=\varphi^{-1}(w\cdot\varphi(\kappa_1(a,b))+(1-w)\cdot\varphi(\kappa_2(a,b))).$$

Clearly, κ_w is a weighted quasi-linear mean of the affinities κ_1 and κ_2 with the weights w and $1-w$, respectively.

Next, the function Δ defined in Def. 6 will have the following form (based on Def. 8):

$$\begin{aligned}\Delta(w) &= sgn(\kappa_w(a,b) - \kappa_w(c,d)) = \\ &= sgn(\varphi^{-1}(w \cdot \varphi(\kappa_1(a,b)) + (1-w) \cdot \varphi(\kappa_2(a,b))) \\ &\quad -\varphi^{-1}(w \cdot \varphi(\kappa_1(c,d)) + (1-w) \cdot \varphi(\kappa_2(c,d)))).\end{aligned}$$

Theorem 3 tells us that the partitions belonging to the 4-tuples have very simple structures in the case of quasi-linear means and this fact plays a crucial role when developing specialized algorithms.

Theorem 3. *Assume that $\langle a,b,c,d\rangle$ is a 4-tuple of spels $(a,b,c,d \in C)$, and let κ_w be an affinity function, as defined in Def. 8. The partition P defined in Def. 7 (belonging to a,b,c,d) satisfies exactly one of the following statements:*
(1) $P = \{[0,1]\}$,
(2) $P = \{\{0\},(0,1]\}$ *or* $P = \{[0,1),\{1\}\}$,
(3) $P = \{[0,w^*),\{w^*\},(w^*,1]\}$ *for a* $w^* \in (0,1)$

Proof. Take the following constants:

$$X := \varphi(\kappa_1(a,b)), Y := \varphi(\kappa_2(a,b)), U := \varphi(\kappa_1(c,d)), V := \varphi(\kappa_2(c,d)).$$

In this case, Δ has the form:

$$\Delta(w) = sgn(\varphi^{-1}(w \cdot X + (1-w) \cdot Y) - \varphi^{-1}(w \cdot U + (1-w) \cdot V)).$$

Let l_1 and l_2 denote the following terms got from Δ:

$$\begin{aligned} l_1(w) &= w \cdot X + (1-w) \cdot Y &&= w \cdot (X-Y) + Y \\ l_2(w) &= w \cdot U + (1-w) \cdot V &&= w \cdot (U-V) + V, \end{aligned}$$

which are linear functions of w (actually they are two lines, if we interpret them on $\mathbb{R}$).
The functions φ and φ^{-1}: $[0,1] \to [0,1]$ are both bijections because they are invertible, and φ is increasing by definition; so φ and φ^{-1} are strictly increasing functions. Therefore the following hold for each $w \in [0,1]$:

$$\begin{array}{lccccccccc} (L1) & l_1(w) < l_2(w) & \Rightarrow & \varphi^{-1}(l_1(w)) & < & \varphi^{-1}(l_2(w)) & \Rightarrow & \Delta(w) & = & -1, \\ (L2) & l_1(w) = l_2(w) & \Rightarrow & \varphi^{-1}(l_1(w)) & = & \varphi^{-1}(l_2(w)) & \Rightarrow & \Delta(w) & = & 0, \\ (L3) & l_1(w) > l_2(w) & \Rightarrow & \varphi^{-1}(l_1(w)) & > & \varphi^{-1}(l_2(w)) & \Rightarrow & \Delta(w) & = & 1. \end{array}$$

In the following, we will show that the statements of the theorem can be derived from the relative position of the two lines l_1 and l_2 (which may be easily verified).

(a) If l_1 and l_2 are *identical* (i.e. $X = U$, $Y = V$), then $\Delta(w) = 0$ for each $w \in [0,1]$, hence $P = \{[0,1]\}$.

(b) If l_1 and l_2 are *not identical, but parallel*, i.e. $X - Y = U - V$ and $Y \neq V$, then $l_1(w) < l_2(w)$ or $l_1(w) > l_2(w)$ on whole $\mathbb{R}$, thus from $L1$ and $L3$, $\Delta(w) = -1$ or $\Delta(w) = 1$ for each $w \in [0,1]$, respectively. In this case, $P = \{[0,1]\}$ once again.

(c) If l_1 and l_2 are *not parallel*, (i.e. $X-Y \neq U-V$), so $X-Y-U+V \neq 0$, then they have an intersection in a given point $w^* \in (-\infty, \infty)$, which can be determined as follows:

$$w^* \cdot X + (1 - w^*) \cdot Y = w^* \cdot U + (1 - w^*) \cdot V,$$

and from here

$$\begin{array}{lcl} w^* \cdot X + (1 - w^*) \cdot Y & = & w^* \cdot U + (1 - w^*) \cdot V \\ w^* \cdot X + Y - w^* \cdot Y & = & w^* \cdot U + V - w^* \cdot V \\ w^* \cdot X - w^* \cdot Y + w^* \cdot V - w^* \cdot U & = & V - Y \end{array}$$

and finally, we can solve it for w^*:

$$w^* = \frac{V - Y}{X - Y + V - U}.$$

Because $X - Y - U + V \neq 0$, w^* is well-defined. There are three cases:

$$\begin{array}{lll} \textbf{(c.1)} & \text{If } w^* \notin [0,1] & \Rightarrow \quad P = \{[0,1]\} \\ \textbf{(c.2)} & \text{If } w^* \in \{0,1\} & \Rightarrow \quad P = \{\{0\},(0,1]\} \text{ or } P = \{[0,1),\{1\}\}, \\ \textbf{(c.3)} & \text{If } w^* \in (0,1) & \Rightarrow \quad P = \{[0,w^*),\{w^*\},(w^*,1]\} \end{array}$$

In the case (c.1), $l_1(w) < l_2(w)$ or $l_1(w) > l_2(w)$ for each $w \in [0,1]$, so $\Delta(w) = -1$ or $\Delta(w) = 1$ are satisfied, as in the parallel case. Thus the whole $[0,1]$ constitutes one equivalence class. The case (c.2) differs from (c.1) in that Δ takes a zero value in 0 or in 1, so there are two equivalence classes (the given endpoint of $[0,1]$ will be a class with one element). In (c.3), there are 3 classes: left from w^*, w^*, and right from w^* according to the relative position of the lines. □

4.2 Algorithms and their complexity

Based on Theorem 3, we can derive new algorithms from Alg. 1 that are specialized for the affinity operators defined in Def. 8.

Our first remark is that the partition refinement step by the interval $[0,1]$ is redundant, because each equivalence class will be replaced by itself (since $[0,1] \cap G_{curr} = G_{curr}$, $[0,1] \setminus G_{curr} = \emptyset$). Hence, if the cases (a), (b), (c.1) occur, the partition refinement step can be skipped. We consider that $[x,x) = (x,x] = \emptyset$ for each $x \in \mathbb{R}$. So we can treat the cases (c.2) and (c.3) together as e.g. $P = \{\{0\},(0,1]\}$ is a special case of (c.3) when $w^* = 0$ and $P = \{[0,0) = \emptyset, \{0\}, (0,1]\} = \{\{0\},(0,1]\}$. Notice that P is clearly defined by the dividing point w^*.

Following the previous statement, we can show that G can be described by $W = \langle w_1, w_2, \ldots, w_k \rangle$ which is the ascending ordered set of the dividing points $w^*_{(1)}, w^*_{(2)}, \ldots, w^*_{(k)}$ corresponding to the iterations of the Alg. 1 in which the cases (c.2) and (c.3) are satisfied. In the first iteration G is partitioned by P_1, so $G^{(1)} = \{[0, w^*_{(1)}), \{w^*_{(1)}\}, (w^*_{(1)}, 1]\}$. In the second iteration there are two cases. If $w^*_{(2)} = w^*_{(1)}$, then $G^{(2)} = G^{(1)}$. If $w^*_{(2)} \neq w^*_{(1)}$, then $w^*_{(2)}$ divides one of the intervals $[0, w^*_{(1)})$ and $(w^*_{(1)}, 1]$ into three parts. For example, let $w^*_{(2)} < w^*_{(1)}$.

Then $[0, w^*_{(1)})$ will be replaced by $[0, w^*_{(2)}), \{w^*_{(2)}\}, (w^*_{(2)}, w^*_{(1)})$. Continuing this, we find that $G = \{[0, w_1), \{w_1\}, (w_1, w_2), \ldots, \{w_k\}, (w_k, 1]\}$, so we can define G by $W = \langle w_1, w_2, \ldots, w_k \rangle$.

The first algorithm specialized for the quasi-linear means can be found in Alg. 2. which was constructed based on our previous observations and Theorem 3. At the start, the set W is initialized. Then the method iterates over all of the possible 4-tuples of spels (steps 2-11). For a given 4-tuple, the constants X, Y, U, V are computed (steps 3-6). In Step 7, the algorithm checks to see whether case (a) or (b) occurs (from Theorem 3), which means that any subsequent computations for that 4-tuple can be skipped (**continue** means that the iteration continues with the next 4-tuple). If the conditions are not satisfied, then the dividing point w^* is computed, and if W does not contain w^*, then W will be augmented by w^* (Step 10). Lastly, in Step 12, the algorithm orders the elements of W, and returns with the dividing points that represent the equivalence classes containing one element, and with the midpoints of the intervals between two dividing points; so it lists the class representatives of G.

Algorithm 2 Naive algorithm for quasi-linear means

Input: $\mathcal{C}$, κ_1, κ_2, φ
Output: the class representatives of G
1: $W \leftarrow \emptyset$
2: **for all** 4-tuple $\langle a, b, c, d \rangle$ **do**
3: $X \leftarrow \varphi(\kappa_1(a, b))$
4: $Y \leftarrow \varphi(\kappa_2(a, b))$
5: $U \leftarrow \varphi(\kappa_1(c, d))$
6: $V \leftarrow \varphi(\kappa_2(c, d))$
7: **if** $X - Y + V - U = 0$ **then continue**
8: $w^* \leftarrow \frac{V-Y}{X-Y+V-U}$
9: **if** $w^* \notin [0, 1]$ **then continue**
10: **if** $w^* \notin W$ **then** $W \leftarrow W \cup w^*$
11: **end for**
12: $\langle w_1, w_2, \ldots, w_k \rangle \leftarrow$ the ascending ordered set of W
13: **return** $\frac{0+w_1}{2}, w_1, \frac{w_1+w_2}{2}, w_2, \ldots, \frac{w_{k-1}+w_k}{2}, w_k, \frac{w_k+1}{2}$

We will now examine the complexity of Alg. 2. We will assume that W is a set implementation where the *add* and *contain* methods require a constant time (e.g. it is a hash set), and the algorithm performs an ordering on the elements of W in Step 12. The advantage of this approach is twofold: 1) if there are many repetitive elements, it costs less if we collect the different elements into an unordered set (with constant adding time) and then we have to sort fewer elements than maintaining an ordered set; 2) if we require just the number of the equivalence classes, we can omit the ordering step. Hence, in the following, we will omit Step 12 from our discussion and we will suppose that it is executed in $\mathcal{O}(|W| \cdot \log(|W|))$ or in $\mathcal{O}(|W|)$

time.

We view one iteration step (steps 3-10) as a constant time operation ($\mathcal{O}(1)$) because the computation of the values X, Y, U, V is always executed and it would be very difficult and time-consuming compared to the other operations (considering the constant time *add* method). Due to the above statements and considerations, the following holds.

Proposition 2. *Regardless the ordering of W, the time complexity of Alg. 2. is $\mathcal{O}(|C|^4)$.*

It is obvious that this complexity is unfeasible for real algorithms. In the following we propose two techniques which singificantly improve its performance.

First, we will assume that each affinity function κ used by our framework satisfies the following. If the spels a and b are not neighbouring, then

$$\kappa(a, b) = 0.$$

Hence, it is sufficient if we consider only the neighbouring pixel pairs and avoid the redundant iterations, so we can modify Alg. 2. (see Alg. 3). The algorithm iterates over all possible pairs of neighbouring pixel pairs and computes w^* (Step 4) as in steps 3-10 in Alg. 2.

Algorithm 3 Algorithm for quasi-linear means - A

Input: $\mathcal{C}$, κ_1, κ_2, φ
Output: the class representatives of G
1: $W \leftarrow \emptyset$
2: **for all** neighbouring pixel pair (a, b) **do**
3: **for all** neighbouring pixel pair (c, d) **do**
4: compute w^* for $\langle a, b, c, d\rangle$ and if it is valid then add it to W
5: **end for**
6: **end for**
7: $\langle w_1, w_2, \ldots, w_k\rangle \leftarrow$ the ascending ordered set of W
8: **return** $\frac{0+w_1}{2}, w_1, \frac{w_1+w_2}{2}, w_2, \ldots, \frac{w_{k-1}+w_k}{2}, w_k, \frac{w_k+1}{2}$

Proposition 3. *Regardless the ordering of W, the time complexity of Alg. 3 is $\mathcal{O}(|C|^2)$.*

Proof. Suppose that each spel has a fixed number of neighbours denoted by k. Then the number of different neighbouring spel pairs is approximately $2k \cdot |C|$, i.e. $\mathcal{O}(|C|)$. Due to the nested for loops, the algorithm executes $\mathcal{O}(|C|^2)$ iterations. □

Note: For the sake of accuracy, if we repeatedly counted the spels which are not neighbours, the algorithm would execute a lot of redundant steps. Both for loops should contain a non-neighbouring pixel pair in order to cover this case exactly once.

Our last approach (Alg. 4.) extends the idea of Alg. 3. If the algorithm computes the same X, Y values (Alg. 2., steps 3-4) for the spel pairs (a_1, b_1), (a_2, b_2), then the pair (a_2, b_2) leads to a sequence of redundant iterations. Alg. 4. tries to avoid this kind of redundancy in such a way that it determines the set of different X, Y pairs for each neighbouring spel pairs (steps 3-7), and it again iterates over this set using two nested loops (steps 8-12).

Algorithm 4 Algorithm for quasi-linear means - B

Input: $\mathcal{C}$, κ_1, κ_2, φ
Output: the class representatives of G
1: $W \leftarrow \emptyset$
2: $S \leftarrow \emptyset$
3: **for all** neighbouring pixel pair (a, b) **do**
4: $\quad X \leftarrow \varphi(\kappa_1(a, b))$
5: $\quad Y \leftarrow \varphi(\kappa_2(a, b))$
6: $\quad$ **if** $(X, Y) \notin S$ **then** $S \leftarrow S \cup (X, Y)$
7: **end for**
8: **for all** $(X, Y) \in S$ **do**
9: $\quad$ **for all** $(U, V) \in S$ **do**
10: $\quad\quad$ compute w^* for $\langle X, Y, U, V \rangle$ and if it is valid then add it to W
11: $\quad$ **end for**
12: **end for**
13: $\langle w_1, w_2, \ldots, w_k \rangle \leftarrow$ the ascending ordered set of W
14: **return** $\frac{0+w_1}{2}, w_1, \frac{w_1+w_2}{2}, w_2, \ldots, \frac{w_{k-1}+w_k}{2}, w_k, \frac{w_k+1}{2}$

Proposition 4. *The time complexity of Alg. 4. regardless of the ordering of* W *is* $\mathcal{O}(|C| + |S|^2)$.

Proof. The determination of the set S (steps 3-7) requires $\mathcal{O}(|C|)$ time because one iteration step contains only a few constant time operations and the number of different neighbouring spel pairs is $\mathcal{O}(|C|)$, can be seen in Prop. 3. The nested loops in steps 8-12 require $\mathcal{O}(|S|^2)$ iterations, so the statement holds. □

We should mention that we can make additional improvements by considering symmetries. When we compute S we can leverage that $\kappa(a, b) = \kappa(b, a)$, so if we iterate over the spels, for a particular a we need to consider only the subsequent spels as bs which clearly halves the number of iterations in the first loop[1]. Also, $(a, b), (c, d) \equiv (c, d), (a, b)$ thus for a particular (X, Y) we need to consider the subsequent (U, V) pairs in the collection[2] which decreases the iteration number in the second loop from $|S|^2$ to $|S|^2/2$.

[1] If the iteration starts from the left upper corner and go from left to right and top to bottom taking 4-connected neighbourhood, for a particular spel we need to consider its right and bottom neighbours.

[2] If S is represented as a set then it has to be converted to an indexed collection, which can usually be done in linear time.

Lastly, we should remark that the algorithm complexities in Propositions 2, 3 and 4 do not rely on the dimensionality of C.

5 Experiments

Although our focus is mainly on theoretical results in this study (namely how we can characterize and determine the equivalence classes belonging to a certain type of affinity operators), we were also interested in testing the given algorithms on real images. Our aim here was to determine how many equivalence classes belong to a particular image and to measure the running times in practice. All the results shown in the following were measured on a PC with a 2 GHz Intel Core i7 CPU and the algoritms were implemented in the Java programming language. In our experiments, two medical image sets were used : 1) 25 digital dermoscopy images of size 1280×1024 pixels, each contains one or more skin lesions, in RGB colour space (Fig. 1) and 2) 3×25 simulated brain MRI slices of size 181×217 (Fig. 2). Simulated T1, dual-echo T2, and proton density PD-weighted slices with 3% noise and 20% inhomogeneity were utilized [14, 15]. As base affinities (κ_1 and κ_2 in Def. 8), a standard homogeneity-based and a standard object feature-based functions were applied [10, 11].

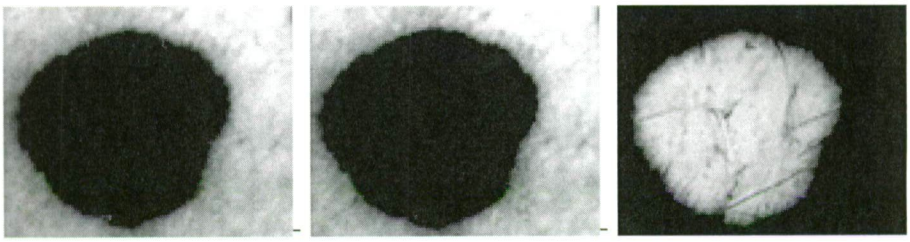


Figure 1: Dermoscopy images (from left to right): grayscale image, blue channel and a special scene based on color difference

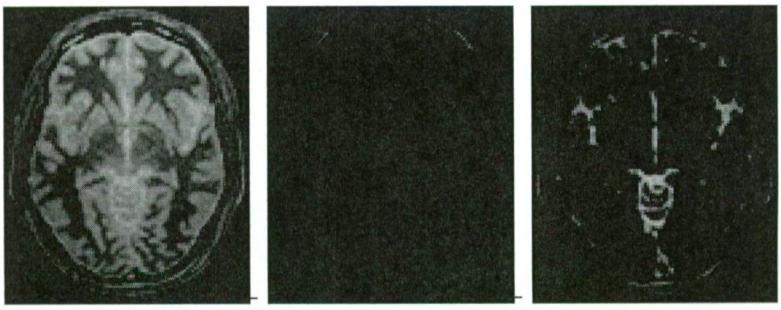


Figure 2: BrainWeb images (from left to right): PD, T1 and dual-echo T2 protocols

Dataset	#	Feature image(s)	Base affinities	Mean
Dermatology	1	grayscale	hom./obj.	geometric
	2	B channel [3]	hom./obj.	arithmetic
	3	a special scene [4]	hom./obj.	arithmetic
BrainWeb	1	T2 and PD	hom.	geometric
	2	T2	hom./obj..	arithmetic
	3	T1	hom./obj.	geometric

Table 1: Test cases for the datasets. The membership scenes (feature images) are extracted according to the methodology of the particular domain. The expressions "hom." and "obj." stand for homogeneity-based and object feature-based affinities, respectively.

We modelled 3 tests each for both datasets, and the results can be seen in Table 1. In each case, a membership scene was extracted according to the methodology of the particular domain, and then we applied Alg. 3 and Alg. 4 to determine the number of equivalence classes and measure the running times. As can be seen in Table 1, in most cases we used an arithmetic mean to aggregate a homogeneity-based affinity with an object feature-based one which is a common combination in the literature. In addition, we provide some examples of using the geometric mean as it is frequently used as well. In the first case of the BrainWeb dataset two homogeneity-based feature images were combined. Clearly, there are many other configurations that can be investigated and our future plan is to create a comprehensive study that deals with this kind of variations, and to compare how the number of affinity classes changes if the image shows the same anatomy but the image acquisition parameters differ or if we fix the acquisition protocol and work with different images the human anatomy.

The results got from our test can be seen in Table 2 and Table 3. Along with the running times and number of iterations, the size of set S (Prop. 4) can be seen as well, which is the number of different (X, Y) pairs computed by Alg. 4. Case columns refer to the test cases defined in Table 1. The majority of the values are averages among the 25 images except the standard deviation of running times.

Testing on both datasets led to an enormous number of equivalence classes (about $10^6 - 10^7$). Alg. 3 is not feasible from a run time perspective, even on the smaller images (BrainWeb sets), while Alg. 4 needs just a few seconds as its improvements drastically reduced the required number of iterations, and Alg. 3 strongly depends on the size of image. The running times do not vary significantly among different images in the same configuration. The number of different (X, Y) pairs (Prop. 4) varies on different images, and does not reflect the image size.

[3] The blue channel in RGB colour space, proposed in [8, 9, 17, 20]

[4] A special membership scene in L*a*b* colour space, where a given spel's membership value reflects its colour distance from the average background colour [37]

BrainWeb		Case-1	Case-2	Case-3
Number of parameters		2.13×10^7	6.16×10^6	1.74×10^6
Alg. 4	Run. time AVG	2.80 s	1.10 s	0.40 s
	Run. time STDEV	0.10 s	0.01 s	0.01 s
	Iterations	5.42×10^7	3.09×10^7	8.24×10^6
	(X, Y) pairs	10411	7857	4057
Alg. 3	Run. time AVG	420.3 s	451.9 s	438.0 s
	Run. time STDEV	7.2 s	2.0 s	9.2 s
	Iterations	1.22×10^{10}	1.22×10^{10}	1.22×10^{10}

Table 2: Results got on BrainWeb datasets. AVG and STDEV denote average and standard deviation, respectively. The expression "(X, Y) pairs" refers to the size of S in Prop. 4, which is an important factor in the time-complexity of Alg. 4.

Dermatology		Case-1	Case-2	Case-3
Number of parameters		4.51×10^6	1.71×10^7	3.48×10^7
Alg. 4	Run. time AVG	1.6 s	3.2 s	6.2 s
	Run. time STDEV	0.4 s	0.3 s	1.2 s
	Iterations	1.16×10^7	4.24×10^7	9.99×10^7
	(X, Y) pairs	4612	8909	14089
Alg. 3	Run. time AVG	$\approx$ 141 h	$\approx$ 141 h	$\approx$ 141 h
	Run. time STDEV	–	–	–
	Iterations	1.37×10^{13}	1.37×10^{13}	1.37×10^{13}

Table 3: Results on dermatology images. AVG and STDEV denote average and standard deviation, respectively. The expression "(X, Y) pairs" refers to the size of S in Prop. 4 which is an important factor in the time-complexity of Alg. 4.

6 Conclusions

The equivalence of affinities is a novel concept and it plays an important role in analyzing affinity functions in the FC framework. It tells us that we should note that different techniques used to defining affinity functions may lead to equivalent affinities, thus making these new constructions unnecessary in a real application, as they only increase redundancy. Apart from the theoretical results, practical considerations can be derived as well. For instance, we could use integer arithmetic-based affinities in performance-sensitive applications.

In this paper, we focused on an example where the affinity operator has a parameter with a real value and it maps different affinity functions for different parameter values. These types of operators are used in a very common scenario when a homogeneity and an object feature-based affinity are combined. We constructed a general algorithm schema which could be a template for algorithms that are able to determine the equivalence classes of affinities according to a given affinity operator.

Based on this template, we defined three algorithms for the example in which the above-mentioned affinities are combined using quasi-linear means. The complexity of these algorithms was also considered, and they were tested using two sets of medical images.

The structure of equivalence classes for quasi-linear means-based operators is quite simple and concise from a mathematical point of view. Furthermore, Alg. 4 required only a few seconds to process an image in our tests. Despite these points, the number of equivalence classes was enormous on the test images ($10^6 - 10^7$), which means we narrowed down the search space from the $[0, 1]$ interval[3] to a finite set of 10^7 elements. However, this value is still too high to explore all the different, non-equivalent affinities in a proper application, even if the experiments are performed in an automatic environment without human supervision.

There are many ways we could continue and improve the results of this study in the future. We did not analyze the relationship between the parameter values of the affinity operator and the corresponding segmentation results. We think that the reasonable number of different segmentations (and affinity functions) for a given image should be closer to $10-100$ than to 10^7, and the set of all non-redundant, non-equivalent affinities could be a good starting point to reduce this number. Other use cases could be also considered, when the parameter of the affinity operator is not the weight of combination. Concrete algorithms built up from this schema will also give us information about the complexity. The general algorithm schema could be extended to multiple variables and proper algorithms could be implemented.

Acknowledgements

The authors are grateful to all anonymous referees whose comments and suggestions have significantly improved our original version of this paper. The authors are also grateful for the images provided by the Department of Dermatology and Allergology of Szeged. This study was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0013).

References

[1] Aczél, J. On mean values. *Bull. Amer. Math. Soc*, 54(39):2–400, 1948.

[2] Aczél, J. and Dhombres, J.G. *Functional equations in several variables*, volume 31. Cambridge University Press, 1989.

[3] Beucher, S. et al. The watershed transformation applied to image segmentation. *SCANNING MICROSCOPY-SUPPLEMENT-*, pages 299–299, 1992.

[3] Obviously, in a proper implementation, we could use a floating point type which has a finite set of values

[4] Boykov, Y., Veksler, O., and Zabih, R. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(11):1222–1239, 2001.

[5] Bustince, H., Barrenechea, E., and Pagola, M. Image thresholding using restricted equivalence functions and maximizing the measures of similarity. *Fuzzy Sets and Systems*, 158(5):496–516, 2007.

[6] Bustince, H., Barrenechea, E., and Pagola, M. Relationship between restricted dissimilarity functions, restricted equivalence functions and normal en-functions: Image thresholding invariant. *Pattern Recognition Letters*, 29(4):525–536, 2008.

[7] Carvalho, B.M., Gau, C.J., Herman, G.T., and Kong, T.Y. Algorithms for fuzzy segmentation. *Pattern Analysis & Applications*, 2(1):73–81, 1999.

[8] Celebi, M.E., Iyatomi, H., Schaefer, G., and Stoecker, W.V. Approximate lesion localization in dermoscopy images. *Skin Research and Technology*, 15(3):314–322, 2009.

[9] Celebi, M.E., Iyatomi, H., Schaefer, G., and Stoecker, W.V. Lesion border detection in dermoscopy images. *Computerized Medical Imaging and Graphics*, 33(2):148–153, 2009.

[10] Ciesielski, K.C. and Udupa, J.K. Affinity functions in fuzzy connectedness-based image segmentation i: Equivalence of affinities. *Computer Vision and Image Understanding*, 114(1):146–154, 2010.

[11] Ciesielski, K.C. and Udupa, J.K. Affinity functions in fuzzy connectedness-based image segmentation ii: Defining and recognizing truly novel affinities. *Computer Vision and Image Understanding*, 114(1):155–166, 2010.

[12] Ciesielski, K.C., Udupa, J.K., Saha, P.K., and Zhuge, Y. Iterative relative fuzzy connectedness for multiple objects with multiple seeds. *Computer Vision and Image Understanding*, 107(3):160–182, 2007.

[13] Ciesielski, Krzysztof. *Set theory for the working mathematician*, volume 39. Cam-bridge University Press, 1997.

[14] Cocosco, C.A., Kollokian, V., Kwan, R.K.S., Pike, G.B., and Evans, A.C. Brainweb: Online interface to a 3d mri simulated brain database. In *NeuroImage*. Citeseer, 1997.

[15] Collins, D.L., Zijdenbos, A.P., Kollokian, V., Sled, J.G., Kabani, N.J., Holmes, C.J., and Evans, A.C. Design and construction of a realistic digital brain phantom. *Medical Imaging, IEEE Transactions on*, 17(3):463–468, 1998.

[16] Cormen, T.H., Leiserson, C.E., and Rivest, R.L. *Introduction to algorithms*. MIT Press, 2009.

[17] Emre Celebi, M., Alp Aslandogan, Y., Stoecker, W.V., Iyatomi, H., Oka, H., and Chen, X. Unsupervised border detection in dermoscopy images. *Skin Research and Technology*, 13(4):454–462, 2007.

[18] Fodor, J. and Roubens, M. *Fuzzy preference modelling and multicriteria decision support*, volume 14. Springer, 1994.

[19] Huang, L.K. and Wang, M.J.J. Image thresholding by minimizing the measures of fuzziness. *Pattern recognition*, 28(1):41–51, 1995.

[20] Iyatomi, H., Oka, H., Saito, M., Miyake, A., Kimoto, M., Yamagami, J., Kobayashi, S., Tanikawa, A., Hagiwara, M., Ogawa, K., et al. Quantitative assessment of tumour extraction from dermoscopy images and evaluation of computer-based extraction methods for an automatic melanoma diagnostic system. *Melanoma Research*, 16(2):183, 2006.

[21] Lei, T., Udupa, J.K., Saha, P.K., and Odhner, D. Artery-vein separation via mra-an image processing approach. *Medical Imaging, IEEE Transactions on*, 20(8):689–703, 2001.

[22] Miki, Y., Grossman, R.I., Udupa, J.K., van Buchem, M.A., Wei, L., Phillips, M.D., Patel, U., McGowan, J.C., and Kolson, D.L. Differences between relapsing-remitting and chronic progressive multiple sclerosis as determined with quantitative mr imaging. *Radiology*, 210(3):769–774, 1999.

[23] Nyúl, L.G., Falcão, A.X., and Udupa, J.K. Fuzzy-connected 3d image segmentation at interactive speeds. *Graphical Models*, 64(5):259–281, 2002.

[24] Paige, R. and Tarjan, R.E. Three partition refinement algorithms. *SIAM Journal on Computing*, 16:973, 1987.

[25] Pednekar, Amol S and Kakadiaris, Ioannis A. Image segmentation based on fuzzy connectedness using dynamic weights. *Image Processing, IEEE Transactions on*, 15(6):1555–1562, 2006.

[26] Pham, D.L. and Prince, J.L. Adaptive fuzzy segmentation of magnetic resonance images. *Medical Imaging, IEEE Transactions on*, 18(9):737–752, 1999.

[27] Preparata, F.P. and Shamos, M.I. Computational geometry: an introduction, 1985. *New York*.

[28] Rice Jr, B.L. and Udupa, J.K. Clutter-free volume rendering for magnetic resonance angiography using fuzzy connectedness. *International Journal of Imaging Systems and Technology*, 11(1):62–70, 2000.

[29] Saha, P.K. and Udupa, J.K. Relative fuzzy connectedness among multiple objects: theory, algorithms, and applications in image segmentation. *Computer Vision and Image Understanding*, 82(1):42–56, 2001.

[30] Saha, P.K., Udupa, J.K., and Odhner, D. Scale-based fuzzy connected image segmentation: theory, algorithms, and validation. *Computer Vision and Image Understanding*, 77(2):145–174, 2000.

[31] Samarasekera, S., Udupa, J.K., Miki, Y., Wei, L., and Grossman, R.I. A new computer-assisted method for the quantification of enhancing lesions in multiple sclerosis. *Journal of computer assisted tomography*, 21(1):145–151, 1997.

[32] Sethian, J.A. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, volume 3. Cambridge university press, 1999.

[33] Tizhoosh, H.R. Image thresholding using type ii fuzzy sets. *Pattern recognition*, 38(12):2363–2372, 2005.

[34] Udupa, J.K. and Saha, P.K. Fuzzy connectedness and image segmentation. *Proceedings of the IEEE*, 91(10):1649–1669, 2003.

[35] Udupa, J.K. and Samarasekera, S. Fuzzy connectedness and object definition: theory, algorithms, and applications in image segmentation. *Graphical Models and Image Processing*, 58(3):246–261, 1996.

[36] Udupa, J.K., Wei, L., Samarasekera, S., Miki, Y., Van Buchem, MA, and Grossman, R.I. Multiple sclerosis lesion quantification using fuzzy-connectedness principles. *Medical Imaging, IEEE Transactions on*, 16(5):598–609, 1997.

[37] Xu, L., Jackowski, M., Goshtasby, A., Roseman, D., Bines, S., Yu, C., Dhawan, A., and Huntley, A. Segmentation of skin cancer images. *Image and Vision Computing*, 17(1):65–74, 1999.

[38] Zhuge, Y., Udupa, J.K., and Saha, P.K. Vectorial scale-based fuzzy-connected image segmentation. *Computer Vision and Image Understanding*, 101(3):177–193, 2006.

Received 28th March 2014

Acta Cybernetica **21** (2014) 629–653.

Database Slicing on Relational Databases*

Dávid Tengeri† and Ferenc Havasi‡

Abstract

Many software systems today use databases to permanently store their data. Testing, bug finding and migration are complex problems in the case of databases that contain many records. Here, our method can speed up these processes if we can select a smaller piece of the database (called a slice) that contains all of the records belonging to the slicing criterion. The slicing criterion might be, for example, a record which gives rise to a bug in the program. Database slicing seeks to select all the records belonging to a specific slicing criterion. Here, we introduce the concept of database slicing and describe the algorithms and data structures necessary for slicing a given database. We define the *Table-based* and the *Record-based slicing* algorithms and we empirically evaluate these methods in two scenarios by applying the slicing to the database of a real-life application and to random generated database content.

Keywords: database theory, program slicing, database slicing

1 Introduction

Program slicing is an interesting topic in software engineering. This concept was first introduced by Weiser [12, 13, 14] and it has grown into a respectable area of research. The technique can be used to select a small part of the program by choosing the relevant statements from the source code. These statements are interesting to the developer from some point of view called the slicing criterion. For example, a variable may contain an incorrect value because of a bug in the program. We can select those statements which are probably responsible for the computation of the incorrect value stored in the variable and examine them in more detail. This type of slicing is called backward slicing. With program slicing, we can also determine which parts of the program must be retested after the modification

*This study was supported by the European Union and co-financed by the European Regional Development Fund within the project TÁMOP-4.2.1/B-09/1/KONV-2010-0005.

†PhD student, Software Engineering Department of the University of Szeged, Hungary, E-mail: dtengeri@inf.u-szeged.hu

‡Assistant lecturer, Software Engineering Department of the University of Szeged, Hungary, E-mail: havasi@inf.u-szeged.hu

of a statement in the program source code. This latter type of slicing is called forward slicing.

The program slicing of database-driven applications introduces new problems. The statements responsible for handling data in the database are often not present in the output of a slicing procedure; however, attempts have been made to extend program slicing with database support [1].

Our research question was how we could define slicing on relational databases, and how useful it is in practice.

We defined *database slicing*, which means selecting a small piece of data from the records or tables stored in the database, where each element of the slice is related directly or indirectly to the slicing criterion and this small piece might be an independent part of the database. The slicing criterion is the *Starting Point of Slicing* and the result is a *database slice*. Database slicing may have other applications. For instance, the slice can be used for migration between two or more databases that have the same structure. Slicing can also improve the process of bug finding and testing by selecting those records which probably give rise to a bug, so the developers can then focus on the more relevant parts of the database. Database slicing can also be useful when we wish to understand the structure of the database or to discover connections between the data stored in separate tables. It can be especially useful when a relevant change is necessary in the structure of database (e.g. change the value of the key column) because slicing can help to determine the impact of this change.

Here, we make the following contributions:

- We define the concept of database slicing and the two different type of slicing algorithms concerning relational databases.

- We test the new algorithms in two different scenarios. The first one was a reproducible environment based on the Drupal CMS and the second one was a real-life application developed at out department and used by more than 40 hospitals daily. We measured the memory consumption and execution time of the algorithms in addition to the selection capability.

This paper is organized as follows. Section 2 contains the required background knowledge needed to understand the concepts introduced later like the basic definitions of database theory and program slicing. In Section 3, we present our definitions of database slicing. In Section 4 we provide a detailed description of slicing algorithms, then to compute a database slice, it is necessary to know the dependencies among the tables. Section 5 we present results of our implementation tested on popular PHP-based content management systems, and a real-life application. After giving an outline of some related articles in Section 6, we round off with a brief summary and suggest some possible future directions of study.

2 Background knowledge

Here, we provide the background knowledge needed to understand the theory of database slicing.

2.1 Database

A lot of today's software uses databases to store their data. In most cases, these databases are relational databases. The general basis of a relational database is the one defined by E.F. Codd in 1970 [4, 5].

We will now describe the data in the relational model via relations. These relations store their data in tables that have columns and rows. Moreover, the model contains relational operators that can be used to handle the data stored in the tables. The following definitions are needed for the relational model ([5]):

Domain: The set of data values of similar types. Let $D_1, D_2, \ldots, D_n$ $((n > 0))$ be domains. Then the $D_1 \times D_2 \times \ldots \times D_n$ set contains all possible n-tuples $\{t_1, t_2, \ldots, t_n\}$, where $t_i \in D_i$ for all i.

Attribute: Instead of indices $(1, 2, \ldots, n)$ used in the previous definition, we can use an unordered set where we store not just the domain, but also the unique identifier of each $t_i \in D_i$ element. Hence the elements of $\{t_1, t_2, \ldots, t_n\}$ will be $(A : v)$ pairs, where A is the attribute of the element and v is a value taken from the domain of A.

Relation: This is a subset of $D_1 \times D_2 \times \ldots \times D_n$, where its elements have the same attributes. Each relation has a tabular representation with the following properties:

1. It does not contain duplicate rows (records).
2. The row order is irrelevant.
3. The attribute (column) order is irrelevant.
4. Each element of the table is atomic; that is, the database management system (DBMS) can store its value directly.

Let $R(A : a, B : b, C : c, \ldots)$ be the relation, where A is an attribute whose domain is a, B is an attribute whose domain is b, etc. Here we can ignore the domains for the sake of clarity - hence we will just use $R(A, B, C, \ldots)$. A relation is a base relation if its elements cannot be derived from any other base relation. The base relations will be the data tables in the DBMS.

Relational schema: The description of the database elements via relations.

Key: Let U be a set of relation attributes. The $U - component$ of $t \in R$ is the set of $(A : v)$ pairs obtained by deleting those pairs from t whose attribute is not in U.

There is a set of candidate keys for each relation. Let K be a candidate key of relation R if K is a subset of the attributes of R. Moreover, we will assume that

1. There are no two rows in R with the same $K - component$.
2. If any of the attributes have been deleted from K, then the above statement (the uniqueness property) will not hold.

Primary key: For each relation, one of the candidate keys is selected as a primary key.

Foreign key: Links between tables are not represented by pointers but data values. The association is a reference to the value of the key.

In [4], a foreign key is an attribute or a set of attributes of R which is not the primary key of R, but contains the value of the primary key of another relation S.

Example 1. Let us take a simple example which contains three tables taken from a website. The *users* table stores the users of our site. The *node* table contains the content sent by the users and the *comments* table stores the comments written to the nodes by the users. The relational schema of the database is:

- users(uid, name, pass, mail)
- node(nid, *uid*, title, body, created)
- comments(cid, *nid*, *uid*, comment, timestamp)

The underlined attributes are the primary keys and attributes in Italics are foreign keys. For example, node.*uid* and comment.*uid* reference the value of users.*uid*.

Example 2. The above-mentioned database in tabular form containing some sample data:

Table 1: Users of the sample website.

users			
uid	name	pass	mail
1	admin	452ce338	admin@localhost
2	shatho	ec80ac93	shatho@localhost
3	sleshec	02f27fcf	sleshec@localhost

Table 2: Content sent by users.

node				
nid	uid	title	body	created
1	2	Zelus Esca	Valde roto vicis metuo cau...	2010-01-21
2	3	Hendrerit	Capto oppeto letalis incass...	2010-02-14
3	1	Humo Cam...	Huic neque tincidunt...	2010-02-17
4	3	Praemitt...	Jumentum nobis si loqu...	2010-03-01
5	2	Si Natu D...	Hos luctus quadrum. M...	2010-03-23

Table 3: Comments of the users.

comments				
cid	nid	uid	comment	timestamp
1	1	2	Tego aliquip exerci nimis nunc torq...	2010-02-01
2	1	2	Luctus nisl populus gilvus consequa...	2010-02-02
3	2	1	Inhibeo abbas imputo patria quae. N...	2010-02-14
4	3	3	Jus ibidem roto pertineo lobortis i...	2010-02-21
5	2	3	Qui roto incassum refoveo uxor sing...	2010-03-01
6	2	1	Exputo mauris pertineo vulpes typic...	2010-03-02
7	5	2	Gemino tamen zelus bene quae jus ca...	2010-03-24
8	4	1	Appellatio esca defui abigo suscipi...	2010-03-25

2.2 Program slicing

Finding and fixing bugs in large and complex software systems is not a trivial task. It is often not easy to determine which parts of the program are affected by a bug.

A *program slice*, introduced by Weiser [12, 13, 14], contains the relevant parts of the source code that are important from a slicing point of view. This view is a statement of the program and this point will be called the *slicing criterion*. The process of determining the program slice is called *program slicing*.

Program slice: The program slice S may be obtained from the program P by dropping statements, so S reproduces a part of the functionality of P.
Hence, the slice is a subset of statements of a program P. The elements of this set participate directly or indirectly in computing the value at the slicing criterion.

Slicing criterion: This is the starting point of slicing. The outcome of program slicing will contain those statements which participate in the computation of the value at the slicing criterion. In general, it has the form (line in source

code,set of variables) or (input values, line in source code,set of variables). The actual form depends on the type of slicing performed.

There are two types of program slicing, namely static slicing and dynamic slicing. Both slicing methods have two directions - forward and backward slicing -, but they have different purposes in each case.

Static slicing: In static slicing, the only input is the source code of the program so we do not have any dynamic information such as the input data of the program or user events. Weiser generates sets of statements in his algorithm. But Ottensetin and Ottenstein outlined another approach in [10]. They defined the program dependence graph (*PDG*) and the computation of a program slice is turned into an accessibility problem.

PDG: This is a directed graph where its vertices are the statements of the program and the edges represent the data and control dependencies between the statements. The slicing criteria is a vertex of the graph and the results of slicing contain the vertices that are accessible via the slicing criterion.

There are two directions of static slicing. The difference between them is the direction of graph traversal.

Backward slicing: With backward slicing, we start traversing the graph in a backward direction from the starting point. This slicing method looks for those statements which take part in computing the value of the slicing criterion. The results of slicing can later be used to find bugs in a program.

Forward slicing: The results of a forward slicing will contain those statements that depend on the value of the slicing criterion. This is useful if we wish to determine which parts of the program will be affected by our modification, so we know from the results which parts of the modified program ought to be retested.

Example 3. Static slicing. This example was taken from [6]. It shows a typical static backward slicing of a program. The slicing criterion was the pair (10,product).

Dynamic slicing: The second type of slicing is dynamic slicing, which works with a specific execution of the program, so it depends on the particular input values and user events. Based on this data we can describe an execution history of the program. A single program statement may appear many times in the execution history and in the slice if it is in a loop or in the body of a function that is called several times. This leads to a more precise solution than that for static slicing, so the results of slicing here contain only those dependencies which take part in the computation of the value of the variable given in the slicing criterion.

	Original program	Result of slicing
(1)	read(n);	read(n);
(2)	i := 1;	i := 1;
(3)	sum := 0;	
(4)	product := 1;	product := 1;
(5)	**while** i <= n **do**	**while** i <= n **do**
	begin	**begin**
(6)	sum := sum + i;	
(7)	product := product * i;	product := product * i;
(8)	i := i + 1;	i := i + 1;
	end;	**end;**
(9)	write(sum);	
(10)	write(product);	write(product);

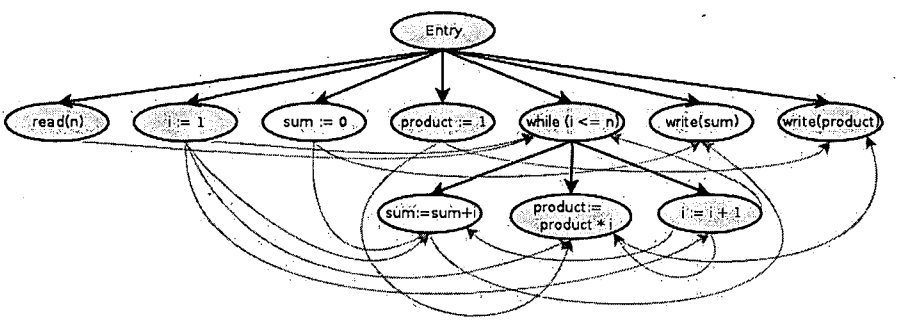


Figure 1: The program dependence graph of the program given in Example 3. The grey vertices are included in the results of the slicing.

Slicing criterion: The slicing criterion now contains an input of the program, so it has the form (x, I^q, V), where x is the input, I in I^q is the line number in the source code, q in I^q is the number of statement in the execution history of the program, and V is a subset of variables.

Backward slicing: With Backward slicing the dynamic case is different from the static case. The direction in the dynamic case will mean the way of computing the slice, not the direction from the statement location. First, we execute and record the execution history. Then we look for the slicing criteria in the history and start the slice backwards from that point. This can produce more precise information for debugging purposes.

Forward slicing: The result can be computed at the time of execution, so this method can save computer resources, as we don't have to store the execution history of the program and start the slicing just after the

running of the program.

DDG: The Dynamic Dependency Graph is defined by Agrawal and Horgen in [7]. This graph contains a node for each statement occurrence in the execution history. The slice will contain the reachable nodes from the starting point.

Example 4. An example of dynamic slicing is taken from [6]. Here, the slicing criterion is (n = 2, 9^{14},z).

		1^1 read(n)	
(1)	read(n);	2^2 i := 1	read(n);
(2)	i := 1;	3^3 i <= n	i := 1;
(3)	**while** i <= n **do**	4^4 (i **mod** 2 = 0)	**while** i <= n **do**
	begin	6^5 x := 18	**begin**
(4)	**if** (i **mod** 2 = 0)	7^6 z := x	**if** (i **mod** 2 = 0)
(5)	x := 17;	8^7 i := i + 1	x := 17;
	else	3^8 i <= n	**else**
(6)	x := 18;	4^9 (i **mod** 2 = 0)	;
(7)	z := x;	5^{10} x := 17	z := x;
(8)	i := i + 1;	7^{11} z := x	i := i + 1;
	end;	8^{12} i := i + 1	**end**;
(9)	write(z);	3^{13} i <= n	write(z);
		9^{14} write(z)	
	Example program	Execution history for input $n = 2$	Result of slicing

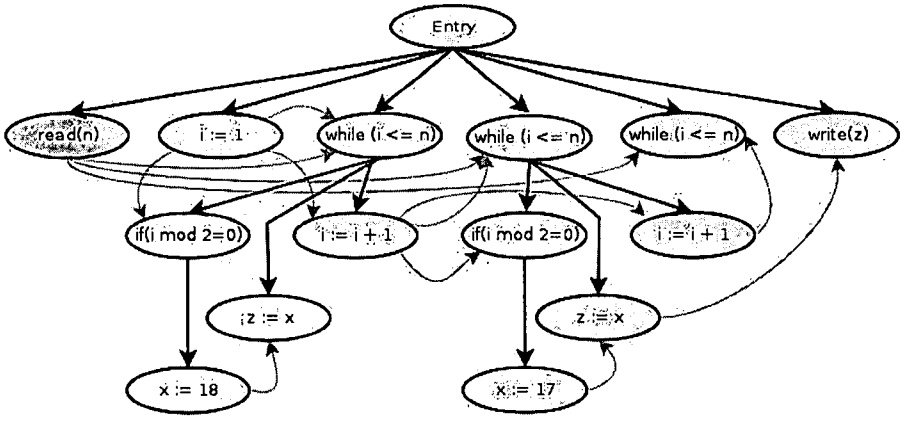


Figure 2: DDG of the program presented in Example 4

Static and dynamic slicing can be combined to compute more relevant slices. For example, in [17] the authors combined static and dynamic information to compute

relevant slices. Relevant slices can be used for regression testing. A relevant slice with respect to a variable occurrence includes all statements that may affect the value of the variable. The relevant slices can be used to determine the test cases in a test suite where the modified program may differ from the original one. The authors in [17] first compute the dynamic slice of the program, then they generate an extended program dependence graph. They add two new dependences between statements of the program called conditional dependence and potential dependence. The relevant slice can be computed by combining information obtained from the extended PDG and the dynamic slice of the program. Their algorithm can be used in a variety of real-life applications.

3 Database slicing

Working with large databases and looking for bugs arising from data stored in a large system or moving data from one database to another raises several issues that need to be addressed:

- Which parts of the database do we need to reproduce a bug?
- Which is the smallest, but still coherent piece of the database that contains the relevant data to reproduce a bug? How should we select this part?
- How can we move just a small piece of the data from one database to another if we want to be sure that each relation of the transported data will exist in the target database at the end of the process? How should we select the relevant records from the database?

Consider the following example based on the database defined in Example 1. When the comments are listed on a page the user name of the second comment has disappeared. If a developer wants to reproduce this bug, she or he has to access the complete database. Perhaps the developer has to copy the database to a local computer. It takes a lot of time if the database contains a lot of data and the developer does not need everything from the database to reproduce the bug. The developer needs just one specific comment and all related content.

Database slicing can help the user to address and handle these issues. Here, we define the key concepts of database slicing.

Database slice: A subset of a database. The slice will contain those database records that have a direct or indirect relation with a given point, which will be taken as the starting point of slicing. A slice may contain dangling references or it may be an independent part of the database.

Database slicing: This is the process of slicing a database. The idea behind database slicing was naturally inspired by program slicing. We incorporated the notion of program slicing into the database context.

Starting point of slicing (SPS): We begin the slicing from this point. The results will include information about those database records or tables that have a direct or indirect connection with this point. Inter-table relations are based on foreign keys in relational databases.

3.1 Types of database slicing

In analogy with program slicing, we will define several kinds of slicing and the directions for database slicing. The first two types depend on the basic elements of the slicing.

Table-based slicing: Table-based slicing works with the tables of a database. It is only aware of the database structure and does not know anything about the data stored in the tables. SPS will be a set of database tables in this case and the results will contain those tables whose records have a direct or indirect relation with each other. Table-based slicing uses only the database schema (structures of the tables and its connections) and it does not depend on the actual content of them. Hence we can say it uses only static information, as is the case with static slicing.

Record-based slicing: Record-based slicing works with records instead of tables. It is also aware of the stored data as well as the structure of the database, so its results can be more precise. Here, the SPS is now a set of records. The results will contain those records (taken from separate tables) which are referred to directly or indirectly via foreign keys. With this type of slicing we can select a smaller piece of data from the database. The record-based slicing generates "dynamic" information (the actual content of the database) to be used, just as it does in dynamic program slicing, to produce a more precise result.

In analogy with program slicing, we can define several directions of slicing. These directions differ in the way we take into account the inter-table relations:

Forward slicing (FS): This slice only contains those tables or records which are connected directly or indirectly by foreign keys. The slice starts from the SPS. This direction tells us the name of each table or records that must be exported if we would like to have a coherent set of data related to the starting point.

Backward slicing (BS): Backward slicing can be used for an analysis to find the potential consequences of a change. A backward slice contains those tables or records which are influenced by a modification made at the starting point. This information can be used to check whether this modification harms the data consistency of the database or not. For example, if we change the date of a comment before the node submission date. Afterwards the data becomes inconsistent. The backward slicing can show us the list of tables or records that are influenced by this change at the comment, so we can modify them too

if necessary. Another good example is when we have to change the primary key of a record and we need to know all the places in the database where the value of the old key is used as a foreign key.

Backward+Forward slicing (B+FS): A B+FS slice is the union of a forward and a backward slice started from the same SPS. This type of slice computes the scope of the SPS.

Backward and Backward+Forward slicing may contain dangling references. To produce an independent piece of the database for these types of slicing too, a forward slicing must be executed starting from every element in the result set.

Example 5. Table-based database slicing example.
Let us use the database example in Example 2. A database slice in the case of $SPS = node$ is:

- **FS**: $\{node, users\}$, because the *uid* column of *node* table is a foreign key for the *uid* column of the *users* table.
- **BS**: $\{node, comment\}$. *nid* of *comments* references the table *node*, so it has to be added to the result set.
- **B+FS**: $\{node, users, comments\}$. The unioin of the FS and BS slice.

Figure 3 shows the results of these slicings.

Example 6. Row-based database slicing example.
Let us apply the database example given in Example 2.
Row-based database slice in the case of $SPS = node$ table 1st row:

- **FS**: {1st row of *node*, 2nd row of *users*}
- **BS**: {1st row of *node*, 1st and 2nd rows of *comments*}
- **B+FS**: {1st row of *node*, 2nd row of *users*, 1st and 2nd rows of *comments*}

Figure 4 shows the results of these slicings.

4 Computing a database slice

We will now define a key component of database slicing, namely the Table Dependence Graph. The slicing algorithms will use this graph as input. The graph can represent the structure of a database, including the relations between its elements.

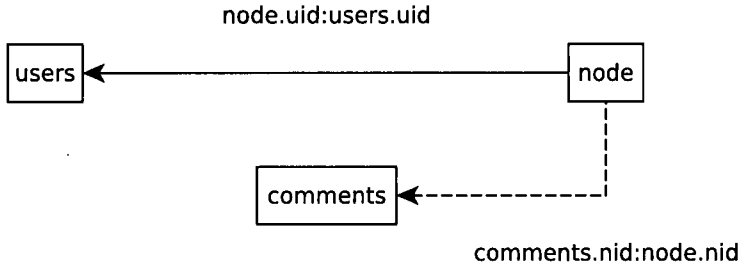
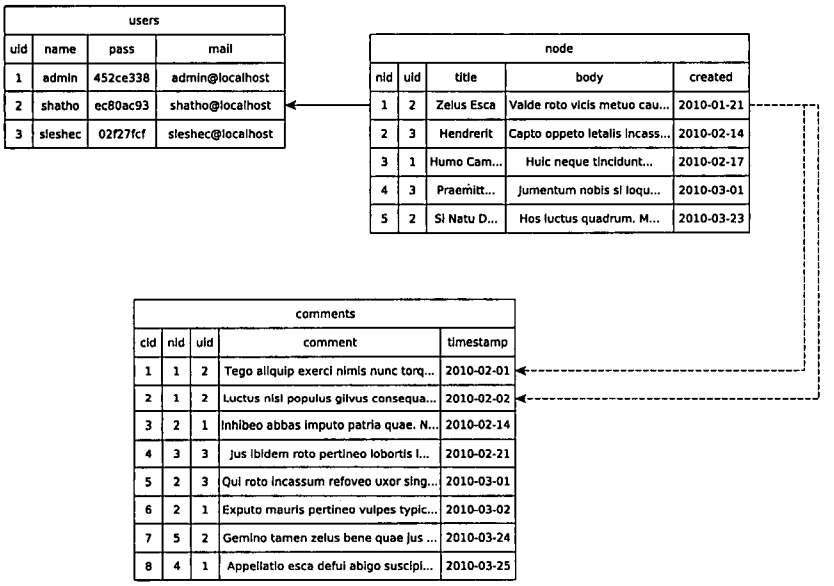


Figure 3: The FS and BS table-based slicing of the database given in Example 5. The result of FS is represented by the solid line and the one with a dashed line represents the result of the BS. The B+FS contains both results.



users

uid	name	pass	mail
1	admin	452ce338	admin@localhost
2	shatho	ec80ac93	shatho@localhost
3	sleshec	02f27fcf	sleshec@localhost

node

nid	uid	title	body	created
1	2	Zelus Esca	Valde roto vicis metuo cau...	2010-01-21
2	3	Hendrerit	Capto oppeto letalis incass...	2010-02-14
3	1	Humo Cam...	Huic neque tincidunt...	2010-02-17
4	3	Praemitt...	Jumentum nobis si loqu...	2010-03-01
5	2	Si Natu D...	Hos luctus quadrum. M...	2010-03-23

comments

cid	nid	uid	comment	timestamp
1	1	2	Tego aliquip exerci nimis nunc torq...	2010-02-01
2	1	2	Luctus nisl populus gilvus consequa...	2010-02-02
3	2	1	Inhibeo abbas imputo patria quae. N...	2010-02-14
4	3	3	Jus ibidem roto pertineo lobortis i...	2010-02-21
5	2	3	Qui roto incassum refoveo uxor sing...	2010-03-01
6	2	1	Exputo mauris pertineo vulpes typic...	2010-03-02
7	5	2	Gemino tamen zelus bene quae jus ...	2010-03-24
8	4	1	Appellatio esca defui abigo suscipi...	2010-03-25

Figure 4: The FS and BS record-based slicing of the database given in Example 6. The solid line represents the results of FS and the dashed lines represent the results of BS. B+FS is the union of these result sets.

4.1 Table-based slicing - Table Dependence Graph

The key part of a slice is the *table dependence graph* (TDG). As we mentioned earlier in the definition of table-based slicing, the results contain the data tables. Hence, the graph will also contain these tables.

Let T be the set of tables and let C_{t_i} be the set of the columns in table t_i for

every $t_i \in T$.

TDG: $TDG = (V, E)$, where

- $V = T$, the set of vertices, the tables in the database.
- E, the set of edges:
  - $E \subseteq T \times T$. Let $t_1, t_2 \in T$. The (t_1, t_2) is an edge in E if there is a $c \in C_{t_1}$ which is a foreign key to one of the rows in t_2.
  - Let the label of this (t_1, t_2) edge be "$t_1.c_1 : t_2.c_2$", where $c_1 \in C_{t_1}$ and $c_2 \in C_{t_2}$, and c_1 is a foreign key to c_2. We call t_1 the *referrer table*, t_2 the *referred table*, c_1 the *referrer column* and c_2 the *referred column*.

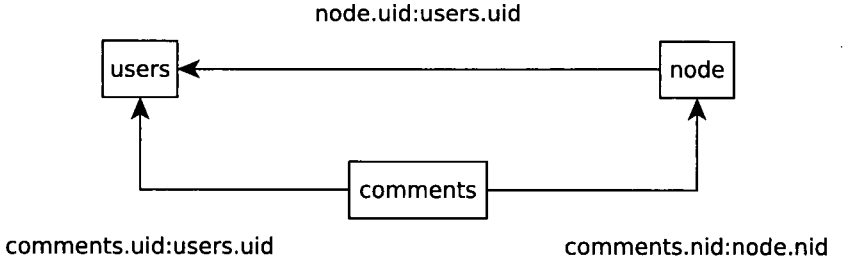


Figure 5: Instance of a table dependence graph based on the database described in Example 2. The vertices of the graph represent the tables.

4.2 Table-based slicing - computing slices

In essence, the slicing algorithm generates the solution of a reachability problem. The key part of the algorithm is the breadth-first search on the TDG. The algorithm requires some input parameters:

- TDG: a table dependence graph.
- SPS: a table of the database, represented by a node of the graph.
- D: the direction of slicing ('FS', 'BS' or 'B+FS').

The algorithm itself is quite simple. We just call $TSLICE\_DO$ with the corresponding parameters, where $TSLICE\_DO$ uses a modified version of the breadth-first search.

The methods $ENQUEUE$, $DEQUEUE$ were taken from [15] and they act on queue data structure.

Figure 6 shows how the table-based slicing works on the graph shown in Figure 5.

Algorithm 1 Method used for table-based slicing

```
// TDG: this table dependence graph will be sliced.
// SPS: starting point of slicing (set of tables)
// D: direction of slicing.
TSLICE(TDG, SPS, D)
1: if D ='B + FS' then
2:     S1 ← TSLICE_DO(TDG, SPS, 'F')
3:     S2 ← TSLICE_DO(TDG, SPS, 'B')
4:     S ← S1 ∪ S2
5: else if D =' FS' then
6:     S ← TSLICE_DO(TDG, SPS, 'F')
7: else
8:     S ← TSLICE_DO(TDG, SPS, 'B')
9: return S
```

4.3 Record-based slicing - computing slices

Record-based slicing takes a slice of the records of a database. SPS is a record of a table in this case. The result set then contains the corresponding rows. The algorithm uses the TDG of the database to discover the connections among the records. The result set contains row references:

Row reference: A (t, K) pair, where

t means the table node that contains the row, and

K means the list of primary key values of the row.

In Algorithm 4, GREYSET and BLACKSET can be stored either in the memory - as we did in *RSLICE_DO* - or they can be stored in a database as well to avoid an overly large memory consumption. In both cases it is sufficient to store just the data present in the row reference - not all the columns of the rows.

5 Experimental results

5.1 Research question to validate

In this section we would like to answer to the following **Research Questions**:

RQ1 How large is the size of the slice (with respect to the size of the database)?

RQ2 How much resources (memory and runtime) are used for computing the slices?

The efficiency of our algorithm is database structure- and data-dependent, so unfortunately we cannot provide a relevant and general answer to these questions. To estimate its efficiency, we looked at two different test scenarios:

Algorithm 2 Table-based slicing algorithm. Modified version of breadth-first search.

```
// TDG: table dependence graph.
// SPS: starting point of slicing (a table node).
// D: direction: 'F' in case of forward or 'B' in case of backward.
TSLICE_DO(TDG, SPS, D)
 1: for ∀u ∈ V[TDG] − {SPS} node do
 2:     color[u] ← WHITE
 3: color[SPS] ← GREY
 4: BLACKSET ← ∅, GREYSET ← ∅
 5: ENQUEUE(GREYSET, SPS)
 6: while GREYSET ≠ ∅ do
 7:     u ←DEQUEUE(GREYSET)
 8:     if D = 'F' then
 9:         E ← {e ∈ E[TDG]|e is an outgoing edge of u}
10:     else
11:         E ← {e ∈ E[TDG]|e is an incoming edge of u}
12:     for ∀e ∈ E do
13:         if D = 'F' then
14:             v ← e.referred_table
15:         else
16:             v ← e.referrer_table
17:         if color[v] = WHITE then
18:             color[v] ← GREY
19:             ENQUEUE(GREYSET, v)
20:     color[u] ← BLACK
21:     BLACKSET ← BLACKSET ∪ u
22: return BLACKSET
```

- For the basis of the first scenario, we chose one of the most popular PHP content management system, namely Drupal, and we generated random data with its *devel* module.

- For the basis of the second scenario, we used a real-life Drupal-based application, namely an information system that is employed in more than 40 hospitals in Hungary - and debugging this system was one of the strongest motivations behind our study.

5.2 Main findings of experiments

The implementation of slicing algorithms was written in Java, and tested on a computer with an Intel Core i5 CPU running at 2.8 GHz and 4GB of RAM.

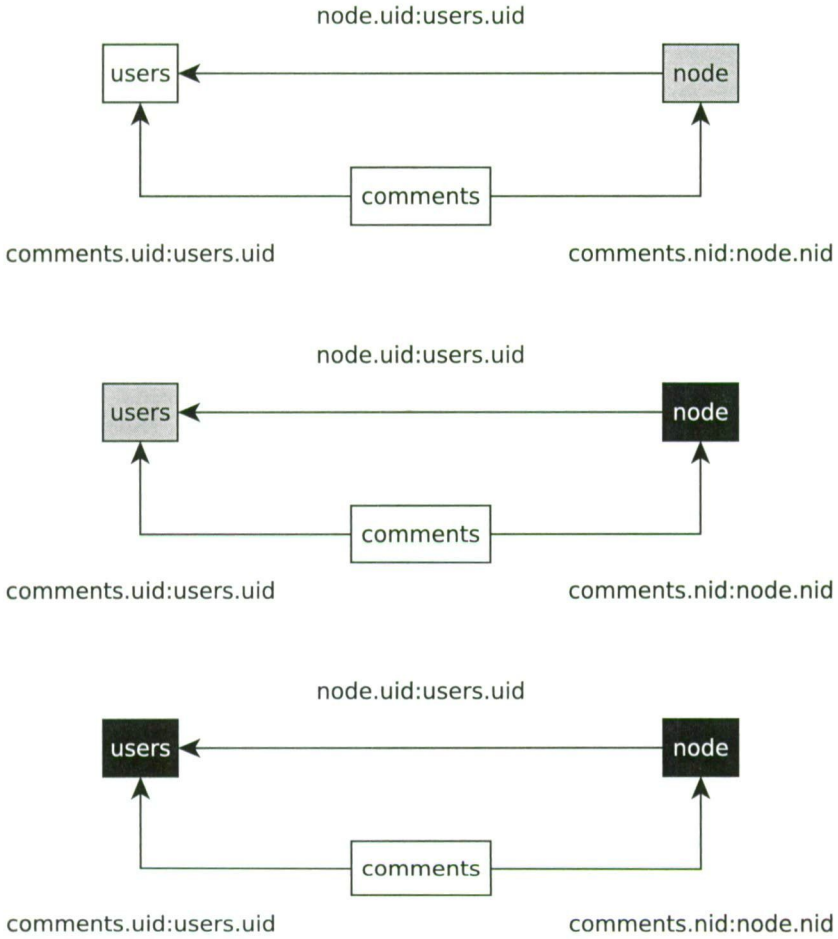


Figure 6: The table-based slicing of a database. The grey nodes have to be visited by the algorithm. The black nodes are those that have been completely processed by the algorithm and are in the result set.

5.2.1 Experiments on random generated data

In our first scenario, we randomly generated the users, the nodes and the comments with the help of *devel* module, which is a powerful helper module for Drupal developers. We used one of its functions (the content generation) to generate random content into the test database on Drupal version 7.15 (base installation with all modules enabled) under the Linux operating system using the Apache Web server and the PostgreSQL database system.

Our test database stored 100 random users and we increased the number of

Algorithm 3 Method used for record-based slicing.

```
// TDG: table dependence graph.
// SPS: starting point of slicing (a row reference).
// D: direction of slicing. (Can be 'FS', 'BS' or 'B+FS'.)
RSLICE(TDG, SPS, D)
 1: if D ='B + FS' then
 2:     S1 ← RSLICE_DO(TDG, SPS,'F')
 3:     S2 ← RSLICE_DO(TDG, SPS,'B')
 4:     S ← S1 ∪ S2
 5: else if D ='F' then
 6:     S ← RSLICE_DO(TDG, SPS,'F')
 7: else
 8:     S ← RSLICE_DO(TDG, SPS,'B')
 9: return S
```

nodes for each measurement. Each node had a maximum of 15 comments that were written by randomly chosen users. We selected the central table of Drupal, the *node* table, as our SPS and executed record-based slicings from this point. We measured the execution time and memory consumption of our algorithm.

Table 4 lists the size of each result set for each direction. The minimum number of nodes in the database was 100 and the maximum was 50,000. We were interested in how the size of the result set varied if the database contained more or fewer records. As can be seen in the table, the forward slice always selected 6 records from the database. The reason is that there are only 6 tables that are reachable from the node table by forward slicing; so the size of result sets will be the same, regardless of the number of records in the database. The size of BS and B+FS increased as the database size got bigger, but the size of the result set was about 1 % or less in every case. In the last scenario, when we generated 50,000 nodes via the devel module, the database contained 1,825,106 records, but the B+FS algorithm selected only 9,954 rows. This is only 0.55 % of the database. The result set size of B+FS slicing is always equal to $|FS| + |BS| - 1$, because the SPS is in both the FS and BS result sets, but it is only counted once for B+FS. The SQL dump of the full database was 2,284 MB. The dump of the result set was 13 MB. In the case of Drupal, as the database increased the number of selected rows did not increase much in terms of the number of records in the database.

Figure 7 shows the execution times and memory consumption for each type of slicing and for the two different types of storage used during the execution of the slicing algorithm. Upon examining the diagrams we see that if we use a database storage, the execution time of the algorithm grows with the size of the result set, but the memory consumption is nearly constant. When we used memory storage, the execution time did not change significantly, but the memory consumption increased dramatically as the result set contains more and more row references. The memory consumption was 2,235.43 KB in the case of B+FS slicing

Algorithm 4 Computes record-based slices.

```
// TDG: table dependence graph.
// SPS: starting point of slicing, a row reference.
// D: direction: 'F' in case of forward or 'B' in case of backward.
RSLICE_DO(TDG, SPS)
 1: BLACKSET ← ∅, GREYSET ← ∅
 2: ENQUEUE(GREYSET, SPS)
 3: while GREYSET ≠ ∅ do
 4:     rf ← DEQUEUE(GREYSET)
 5:     t ← rf.t // table node belongs to row_ref rf
 6:     // load row from database belongs to rf
 7:     // value of the columns will be accessible by row[column name]
 8:     row ← DB_LOAD_BY_ROW_REF(rf)
 9:     if D = 'F' then
10:         E ← {e ∈ E[TDG]|e is an outgoing edge of t}
11:     else
12:         E ← {e ∈ E[TDG]|e is an incoming edge of t}
13:     for ∀e ∈ E do
14:         if D = 'F' then
15:             rows ← DB_LOAD(e.referred_table,
                        e.referred_column,
                        row[e.referrer_column])
16:         else
17:             rows ← DB_LOAD(e.referrer_table,
                        e.referrer_column,
                        row[e.referred_column])
18:         while ∀r ∈ rows do
19:             rowref ← row reference of r
20:             if NOT (IS_IN(GREYSET ∪ BLACKSET, rowref)) then
21:                 ENQUEUE(GREYSET, rowref)
22:     // move rf from GREYSET to BLACKSET
23:     MOVE(GREYSET, BLACKSET, rf)
24: return BLACKSET
```

on the biggest database using memory storage. The algorithm using database storage required approximately 600 KB to produce the results in each case. As we can see, the memory consumption can be reduced by using a database as a storage, but in our measurements we only managed to save 1,600 KB of memory, which is not so significant. But the execution time of the algorithm that used memory storage, was much faster. It was only 0.91 second, while the one with database storage was 25.2 seconds. So to slice the Drupal database, the memory-based algorithm is more useful in practice.

	Results set size			
Generated content (number of nodes)	FS	BS	B+FS	Records in DB
100	6	55	60	5,846
250	6	113	118	9,260
500	6	140	145	20,031
1,000	6	265	270	36,497
2,500	6	696	701	53,180
5,000	6	1,433	1,438	150,198
10,000	6	2 536	2,541	351,118
50,000	6	9,949	9,954	1,825,106

Table 4: Size of the result set based on the generated content and the type of slicing. FS: Forward slicing, BS: Backward slicing, B+FS: Backward + Forward slicing.

5.2.2 Experiments on real-life application

In the second scenario, instead of using a randomly generated content, we found a real-life application and database to test our algorithm. This system evaluated the blood samples (mb_sample) of children (mb_child) with measurements (mb_result) got with different instruments (mb_machine). Each sample was sampled by a user (mb_employee and users) in an Institute (mb_institute). The system stores some extra information about foreign children (mb_child_foreign).

Figure 8 shows the TDG of the application and Table 5 shows the size of each tables. The total number of records stored in the database was 1,270,463.

Table	Number of records
mb_sample	216,870
mb_result	861,027
mb_child	191,988
mb_employee	260
mb_institute	47
mb_machine	10
users	261

Table 5: Size of the tables in the real life application

As a starting point, we selected different records from different tables. Table 6 contains the results of set sizes of B+FS slicing starting from different tables.

The results are shown in Figure 9. We got similar results (for the execution time and memory consumption) to those for randomly generated content. There is a curious thing in Figure 9a: the execution time of B+FS slicing - with memory storage - starting from the mb_machine table produces the largest result set, but it is much faster than those having a result set that is ten times smaller. The

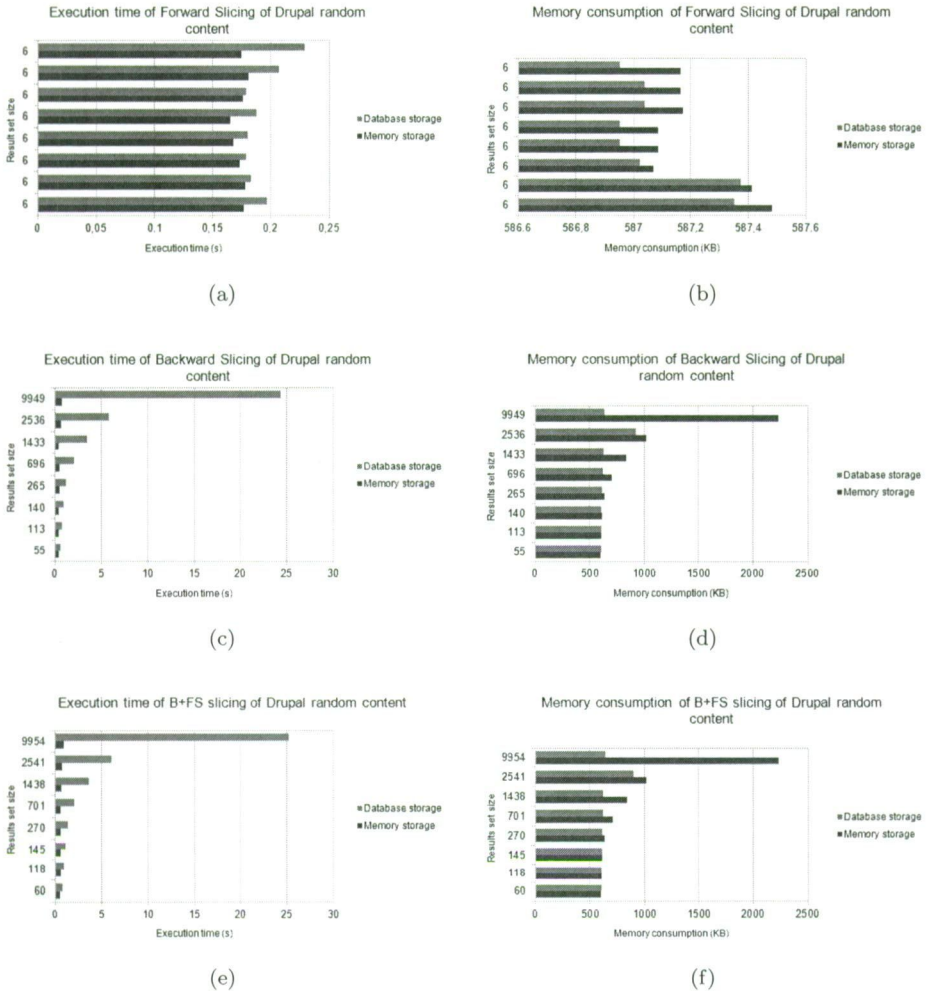


Figure 7: The execution times and memory consumption for the record-based slicing algorithm in the case of randomly generated contents.

reason for this seems to be that the algorithm has to visit only one table and also the starting table to collect results. This table is the mb_result and the algorithm cannot step backwards from this point. The maximum selection size was 16.35 % when the SPS was in the mb_machine table. Also, the number of records in the rest of the result sets was under 1 % compared to the database sizes.

In summary, the result sets of the database slicing contained less than 1% of the records of the entire database in most of the cases and the maximum selection size was 16.35%. The resource requirements of the algorithm was sufficient in each

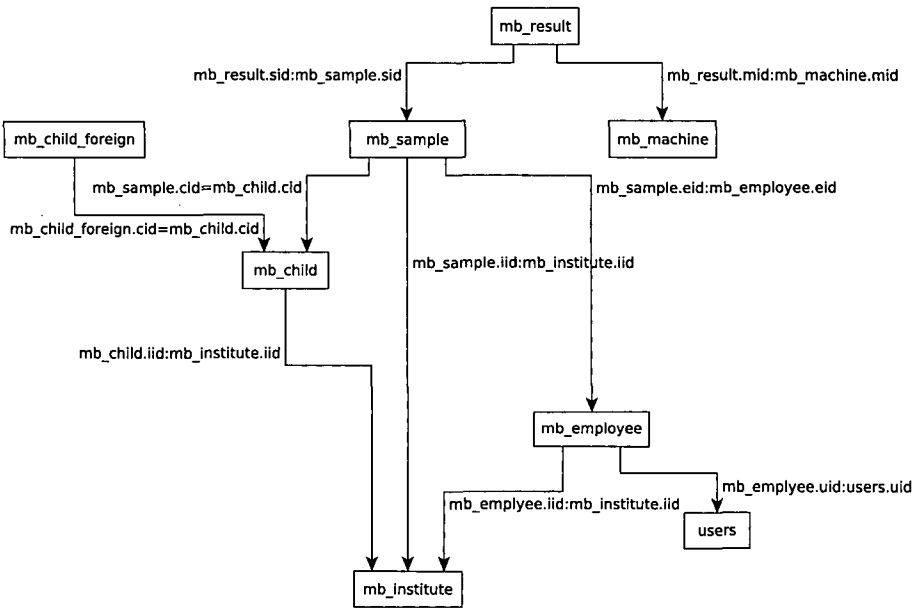


Figure 8: Table Dependence Graph of our real-life application.

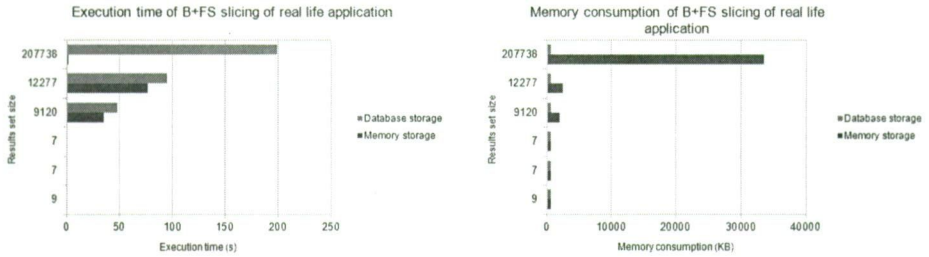
SPS	Results set size
mb_sample	9
mb_result	7
mb_child	7
mb_employee	9,120
mb_institute	12,277
mb_machine	207,738

Table 6: The size of the result set based on the different starting tables.

case.

5.3 Threats to validity

Unfortunately, it cannot be guaranteed, that these kinds of good results can be achieved in our experiments with every application, because the algorithm is database structure- and data-dependent. Hence precise efficiency can vary in practice. The random generation of content in our experiments may not represent the content distribution of a real Drupal-based website, but it involves the most used parts of the Drupal database structure, so it should provide a good picture of how the slicing works in this environment.



(a) (b)

Figure 9: The execution time and memory consumption for the B+FS slicing of our real-life application.

We investigated only Drupal-based applications in our experiments, but our slicing method can be used on every type of application that uses a relational database as its storage because the references among records are defined in the same way as in the case of Drupal.

6 Related work

In Microsoft's patent [8], there is a definition of *slicing of relational databases*, but here they use this concept only in a specific sense. The method they defined is equivalent to our forward record slicing algorithm, and they use it for test data generation. The result of their method is a database that stores a subset of real data taken from the database. They represent the database as a connected graph, based on the schema description of the database. This graph is then used to direct the slicing method and it helps to discover dependencies among records. Their graph is similar to our TDG. It represents the same information as the TDG in our definition. Based on this graph, their algorithm copies the corresponding record from the connected tables too. Their algorithm generates the same results for one record as our forward record-based slicing.

In [16], the authors process the SQL statements of a program to discover interesting inclusion dependencies in the database. Their method works on the execution plans of the SQL queries instead of the raw queries. The execution plan of a query is a description of the steps that the database engine has to perform to retrieve the selected data from the database. The authors define the *NavLog* algorithm which can be used to determine the links between attributes of different relational schemas. This technique can help one to discover the connections among database tables, which we treated as an input to our methods, just as [9].

In [9], the authors provide another approach to finding dependencies in a database. They compare the results of applying reverse engineering techniques to elicit the dependencies among records. The reverse engineering techniques use the

source code as input and for handling database-driven applications. This process is called Database Reverse Engineering (DBRE). One of the main tasks of DBRE is to find the data dependencies among fields. The authors found that these dependencies can be discovered in the source code by looking for those places where the records get stored or modified. Program understanding techniques can be used to find these places in the source code.

Program slicing inspired new techniques in other areas, such as model slicing and model transformation slicing.

Model slicing tries to solve problems arising from the large size and complexity of models like UML models of an industrial application. In [11], the authors describe techniques for the slicing of UML models. The slices are performed using model transformations. The transformed models contain only those parts which specify the properties of a subset of the elements of the original model. The authors' work is based on the Object Constraint Language (OCL) of UML, which can be used to describe rules that apply to models. The authors describe how to slice a model based on the constraints applied to the classes and introduce state machine slicing.

In [18], the author defined a general model-based slicing framework that can be used to slice models that meet the necessary requirements. In his article, he demonstrates the framework by slicing a simple UML model. The author introduces a model definition to describe the meaning of any model and demonstrates slicing using this new model. He describes four types of slicing; namely, static slicing, dynamic slicing, conditioned slicing and slicing unions.

In [19], the authors introduce a dynamic backward slicing technique for model transformation programs and the underlying models of the transformation. Their technique relies upon an execution trace of the model transformation program and simultaneously produces slices for the statements of the transformation program and the model elements which affect the slicing criterion.

7 Summary and Future Work

Our research goal was to find out the applicability and usefulness of defining slicing on relational databases. In this paper we presented the following contributions. We introduced the concept of database slicing and its variants: table and record-based slicing, and forward, backward and forward+backward slicing. We provided table-based and record-based slicing algorithms and we evaluated the results using Drupal and a real-life Web-based information system. We learned that it can be useful in practice to help migration, debugging, testing and understanding the software by reducing the size of the database when the dependencies are known, but the result depends on the database structure and the data stored in the tables.

All of the techniques described here just concentrated on the database structures, connections and the corresponding operations and this introduced limitations. We reached a point where it was impossible to significantly improve these slicing techniques by just staying in the database.

In the future, we would like to combine database slicing with the results of

program slicing and analysis, just like those in [9]. Another example is given in [1], where the classical PDG-based slicing technique was extended with the capability of handling database operations. However, it should be optimized and could be made more realistic if the dependencies in the database were properly analysed.

References

[1] David Willmor, Suzanne M. Embury, Jianhua Shao, *Program Slicing in the Presence of Database State.* In Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pages 448-452. IEEE Computer Society, 2004

[2] Devel module, URL: http://drupal.org/project/devel

[3] Drupal, URL: http://drupal.org/ 2002.

[4] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 13(6):377-387, June, 1970.

[5] E. F. Codd, *Extending the Database relational Model to Capture More Meaning.* ACM Transactioins on Database Systems, 4(4):397-434, December 1979

[6] Frank Tip, *A Survey of Program Slicing Techniques.* Journal of Programming Languages 3, pages 121-189, 1995.

[7] H. Agrawal and J. R. Horgan, *Dynamic Program Slicing.* In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 25(6):246-256, 1990.

[8] Hui Shi, Kenton Gidewall, Marcelo M. De Barros, Chan Chaiyochlarb, Murali R. Krishnan, Robert Irwin Voightmann, Christina Ruth Dhanaraj, *Slicing of relational databases.* Patent. US 7873598. April 2008. URL: http://www.google.com/patents/US7873598

[9] Jean Henrard, Jean-Luc Hainaut, *Database dependency elicitation in database reverse engineering.* In CSMR '01: Proceedings of the Conference on Software Maintenance and Reengineering, pages 11-19. IEEE Computer Society, 2002.

[10] K.J. Ottenstein and L.M. Ottenstein, *The program dependence graph in a software development environment.* In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 177-184, 1984.

[11] K. Lano, S. Kolahdouz-Rahimi, *Slicing of UML Models*

[12] M. Weiser, *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigen, Ann Arbor, 1979.

[13] M. Weiser, *Programmers use slices when debugging.* Communications of the ACM, 25(7):446-452, 1982.

[14] M. Weiser, *Program slicing.* IEEE Transactions on Software Engineering, 10(4):352-357, 1984.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill, 2001.

[16] Stéphane Lopes, Jean-Marc Petit, Farouk Toumani, *Discovering interesting inclusion dependencies: application to logical database tuning.* Information Systems, Volume 27, Issue 1, March 2002, Pages 1-19, ISSN 0306-4379, 10.1016/S0306-4379(01)00027-8.

[17] Tibor Gyimóthy, Árpád Beszédes, and István Forgács, *An efficient relevant slicing method for debugging.* In Lecture Notes in Computer Science, 1687:303–321. Springer, 1999.

[18] Tony Clark, *A general model-based slicing framework.* Proceedings of the Workshop on Composition and Evolution of Model Transformations, 2011

[19] Z. Ujhely, Á. Horváth, D. Varró, *Dynamic Backward Slicing of Model Transformations.* International Conference on Software Testing and Validation:1-10 2012.

Received 10th April 2012

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János